# Red Hat build of Node.js 14

## Node.js ランタイムガイド

Node.js 14 を使用して、OpenShift とスタンドアロンの RHEL で実行されるスケーラブルなネットワークアプリケーションを開発します。

Last Updated: 2023-05-12

# Red Hat build of Node.js 14 Node.js ランタイムガイド

Node.js 14 を使用して、OpenShift とスタンドアロンの RHEL で実行されるスケーラブルなネットワークアプリケーションを開発します。

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

## 法律上の通知



## 概要

本ガイドでは、Node.js ランタイムの使用方法について説明します。

# 目次

# はじめに

本ガイドでは、概念と、開発者が Node.js ランタイムを使用するために必要な実用的な詳細情報を説明します。

# 第1章 NODE.JS でのアプリケーション開発の概要

本セクションでは、Red Hat ランタイムでのアプリケーション開発の基本概念を説明します。また、Node.js ランタイムの概要についても説明します。

## 1.1. RED HAT RUNTIMES でのアプリケーション開発の概要

Red Hat OpenShift は、クラウドネイティブランタイムのコレクションを提供するコンテナーアプリケーションプラットフォームです。ランタイムを使用して、OpenShift で Java または JavaScript アプリケーションを開発、ビルド、およびデプロイできます。

Red Hat Runtimes for OpenShift を使用したアプリケーション開発には、以下が含まれます。

- OpenShift で実行するように設計された Eclipse Vert.x、Thorntail、Spring Boot などのランタイムのコレクション。
- OpenShift でのクラウドネイティブ開発への規定的なアプローチ。

OpenShift は、アプリケーションのデプロイメントおよびモニターリングの管理、保護、自動化に役立ちます。ビジネス上の問題を小規模なマイクロサービスに分割し、OpenShift を使用してマイクロサービスをデプロイし、監視し、維持することができます。サーキットブレーカー、ヘルスチェック、サービス検出などのパターンをアプリケーションに実装できます。

クラウドネイティブな開発は、クラウドコンピューティングを最大限に活用します。

以下でアプリケーションをビルドし、デプロイし、管理できます。

**OpenShift Container Platform**

Red Hat のプライベートオンプレミスクラウド。

**Red Hat Container Development Kit (Minishift)**

ローカルマシンにインストールおよび実行できるローカルクラウド。この機能は、Red Hat Container Development Kit (CDK) または Minishift で提供されます。

**Red Hat CodeReady Studio**

アプリケーションの開発、テスト、およびデプロイを行う統合開発環境 (IDE)。

アプリケーション開発を開始できるようにするため、サンプルアプリケーションですべてのランタイムが利用可能になります。これらのサンプルアプリケーションは Developer Launcher からアクセスできます。サンプルをテンプレートとして使用してアプリケーションを作成することができます。

本ガイドでは、Node.js ランタイムに関する詳細情報を提供します。その他のランタイムの詳細は、関連する ランタイムドキュメント を参照してください。

## 1.2. DEVELOPER LAUNCHER を使用した RED HAT OPENSHIFT でのアプリケーション開発

Developer Launcher (developers.redhat.com/launch) を使用して、OpenShift でのクラウドネイティブアプリケーションの開発を開始することができます。これは、Red Hat が提供するサービスです。

Developer Launcher はスタンドアロンのプロジェクトジェネレーターです。これを使用して、OpenShift Container Platform、Minishift、CDK などの OpenShift インスタンスでアプリケーションをビルドし、デプロイできます。

## 1.3. NODE.JS の概要

Node.js は、Google からの V8 JavaScript エンジン をベースとしており、サーバー側の JavaScript アプリケーションを作成できます。効率的なアプリケーションの作成を可能にするイベントおよび非ブロッキング操作をベースとした I/O モデルを提供します。Node.js は、npm と呼ばれる大規模なモジュールエコシステムも提供します。Node.js の詳細は、関連情報 を参照してください。

Node.js ランタイムを使用すると、ローリング更新、継続的デリバリーパイプライン、サービス検出、カナリアデプロイメントなどの OpenShift プラットフォームの利点と利便性を提供しながら、Node.js アプリケーションとサービスを OpenShift で実行できます。また、OpenShift を使用すると、外部化設定、ヘルスチェック、サーキットブレーカー、フェイルオーバーなどの一般的なマイクロサービスパターンをアプリケーションに実装することが容易になります。

Red Hat は、Node.js のさまざまなサポート対象リリースを提供しています。サポートの利用方法の詳細は、Getting Node.js and support from Red Hat を参照してください。

### 1.3.1. Node.js でサポートされるアーキテクチャー

Node.js は以下のアーキテクチャーをサポートします。

- x86_64 (AMD64)
- OpenShift 環境の IBM Z (s390x)
- OpenShift 環境の IBM Power System (ppc64le)

アーキテクチャーによって異なるイメージがサポートされています。本ガイドのコード例は、x86_64 アーキテクチャーのコマンドを示しています。他のアーキテクチャーを使用している場合は、コマンドに該当するイメージ名を指定します。

### 1.3.2. サンプルアプリケーションの概要

クラウドネイティブのアプリケーションおよびサービスをビルドする方法を実証する作業アプリケーションがあります。これらは、アプリケーションの開発時に使用する必要のある規範的なアーキテクチャー、設計パターン、ツール、およびベストプラクティスを示しています。サンプルアプリケーションは、クラウドネイティブのマイクロサービスを作成するためのテンプレートとして使用できます。本ガイドで説明しているデプロイメントプロセスを使用して、これらの例を更新および再デプロイできます。

この例では、以下のような マイクロサービスパターン を実装します。

- REST API の作成
- データベースの相互運用
- ヘルスチェックパターンの実装
- アプリケーションの設定を外部化して、アプリケーションをより安全で拡張しやすくする

サンプルアプリケーションは以下のように使用できます。

- テクノロジーの実用的なデモンストレーション
- プロジェクトのアプリケーションを開発する方法を理解するための学習ツールまたはサンドボックス

- 独自のユースケースを更新または拡張するためのヒント

各サンプルアプリケーションは 1 つ以上のランタイムに実装されます。たとえば、REST API Level 0 のサンプルは以下のランタイムで利用できます。

- Node.js
- Spring Boot
- Eclipse Vert.x
- Thorntail

以降のセクションでは、Node.js ランタイムに実装されたサンプルアプリケーションについて説明しています。

すべてのサンプルアプリケーションを以下にダウンロードおよびデプロイできます。

- x86_64 アーキテクチャー - 本ガイドのサンプルアプリケーションでは、サンプルアプリケーションを x86_64 アーキテクチャーにビルドおよびデプロイする方法を説明します。
- s390x アーキテクチャー - IBM Z インフラストラクチャーでプロビジョニングされた OpenShift 環境にサンプルアプリケーションをデプロイするには、コマンドで関連する IBM Z イメージ名を指定します。
- ppc64le アーキテクチャー - IBM Power System インフラストラクチャーでプロビジョニングされている OpenShift 環境でサンプルアプリケーションをデプロイするには、コマンドに関連する IBM Power System のイメージ名を指定します。
  サンプルアプリケーションの一部には、ワークフローを実証するために Red Hat Data Grid などの他の製品も必要になります。この場合は、これらの製品のイメージ名を、サンプルアプリケーションの YAML ファイルで関連する IBM Z または IBM Power System のイメージ名に変更する必要もあります。

# 第2章 DEVELOPER LAUNCHER を使用したアプリケーションのダウンロードおよびデプロイ

このセクションでは、ランタイムで提供されるサンプルアプリケーションをダウンロードおよびデプロイする方法を説明します。アプリケーションのサンプルは Developer Launcher で利用できます。

## 2.1. DEVELOPER LAUNCHER の使用

Developer Launcher (developers.redhat.com/launch) は OpenShift 上で実行します。サンプルアプリケーションをデプロイする場合、Developer Launcher は以下のプロセスを説明します。

- ランタイムの選択
- アプリケーションのビルドおよび実行

選択に基づいて、Developer Launcher はカスタムプロジェクトを生成します。プロジェクトの ZIP バージョンをダウンロードするか、OpenShift Online インスタンスでアプリケーションを直接起動できます。

Developer Launcher を使用してアプリケーションを OpenShift にデプロイする場合は、Source-to-Image (S2I) ビルドプロセスが使用されます。このビルドプロセスは、OpenShift でアプリケーションを実行するのに必要なすべての設定、ビルド、およびデプロイメントのステップを処理します。

## 2.2. DEVELOPER LAUNCHER を使用したサンプルアプリケーションのダウンロード

Red Hat は、Node.js ランタイムを使い始める際に役立つサンプルアプリケーションを提供します。これらの例は、Developer Launcher (developers.redhat.com/launch) で利用できます。

サンプルアプリケーションをダウンロードして、ビルドして、デプロイできます。本セクションでは、サンプルアプリケーションをダウンロードする方法を説明します。

サンプルアプリケーションをテンプレートとして使用し、独自のクラウドネイティブアプリケーションを作成できます。

**手順**

1. Developer Launcher (developers.redhat.com/launch) に移動します。
2. **Start** をクリックします。
3. **Deploy an Example Application**をクリックします。
4. **Select an Example** をクリックし、ランタイムで使用できるサンプルアプリケーションの一覧を表示します。
5. ランタイムを選択します。
6. サンプルアプリケーションを選択します。

**注記**

複数のランタイムで利用できるサンプルアプリケーションもあります。前の手順でランタイムを選択していない場合は、サンプルアプリケーションで利用できるランタイムの一覧からランタイムを選択できます。

7. ランタイムのリリースバージョンを選択します。ランタイムに一覧表示されているコミュニティーまたは製品リリースから選択できます。
8. **Save** をクリックします。
9. **Download** をクリックして、サンプルアプリケーションをダウンロードします。
ソースおよびドキュメントファイルを含む ZIP ファイルがダウンロードされます。

## 2.3. OPENSHIFT CONTAINER PLATFORM または CDK (MINISHIFT) へのサンプルアプリケーションのデプロイ

サンプルアプリケーションを OpenShift Container Platform または CDK (Minishift) のいずれかにデプロイできます。アプリケーションをデプロイする場所に応じて、認証に該当する Web コンソールを使用します。

**前提条件**

- Developer Launcher を使用して、サンプルアプリケーションプロジェクトを作成している。
- アプリケーションを OpenShift Container Platform にデプロイする場合は、OpenShift Container Platform Web コンソールにアクセスできる必要があります。
- CDK (Minishift) にアプリケーションをデプロイする場合は、CDK (Minishift) Web コンソールにアクセスできるようにする必要があります。
- **oc** コマンドラインクライアントがインストールされている。

**手順**

1. サンプルアプリケーションをダウンロードします。
2. **oc** コマンドラインクライアントを使用して、サンプルアプリケーションを OpenShift Container Platform または CDK (Minishift) にデプロイできます。
Web コンソールによって提供されるトークンを使用してクライアントを認証する必要があります。アプリケーションをデプロイする場所に応じて、OpenShift Container Platform Web コンソールまたは CDK (Minishift) Web コンソールを使用します。以下の手順を実行してクライアントの認証を取得します。
   a. Web コンソールにログインします。
   b. Web コンソールの右上隅にあるクエスチョンマークアイコンをクリックします。
   c. 一覧から **Command Line Tools** を選択します。
   d. **oc login** コマンドをコピーします。
   e. ターミナルにコマンドを貼り付け、お使いのアカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

3. ZIP ファイルの内容を展開します。

```
$ unzip MY_APPLICATION_NAME.zip
```

4. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

5. **MY_APPLICATION_NAME** の root ディレクトリーに移動します。

6. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

注記: アプリケーションの例によっては、追加の設定が必要になる場合があります。サンプルアプリケーションをビルドおよびデプロイするには、**README** ファイルに記載されている手順に従います。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa             1/1       Running     0          58s
MY_APP_NAME-s2i-1-build         0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod は完全にデプロイされて起動すると **Running** ステータスになります。アプリケーションの Pod 名は異なる場合があります。Pod 名の数値は、新規ビルドごとに増分します。末尾の文字は、Pod の作成時に生成されます。

8. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報は、Pod へのアクセスに使用できるベース URL を提供します。この例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスできます。

# 第3章 NODE.JS アプリケーションの開発およびデプロイ

例を使用すること に加えて、新しい Node.js アプリケーションをゼロから作成し、それらを OpenShift にデプロイできます。

## 3.1. NODE.JS アプリケーションの開発

基本的な Node.js アプリケーションの場合は、Node.js メソッドを含む JavaScript ファイルを作成する必要があります。

**前提条件**

- **npm** がインストールされている。

**手順**

1. 新しいディレクトリー **myApp** を作成し、そのディレクトリーに移動します。

```
$ mkdir myApp
$ cd MyApp
```

これは、アプリケーションのルートディレクトリーです。

2. **npm** でアプリケーションを初期化します。
この例の残りの部分では、エントリーポイントが **app.js** であると想定しています。これは、**npm init** の実行時に設定するように求められます。

```
$ cd myApp
$ npm init
```

3. **app.js** という名前の新規ファイルにエントリーポイントを作成します。

**例: app.js**

```
const http = require('http');

const server = http.createServer((request, response) => {
  response.statusCode = 200;
  response.setHeader('Content-Type', 'application/json');

  const greeting = {content: 'Hello, World!'};

  response.write(JSON.stringify(greeting));
  response.end();
});

server.listen(8080, () => {
  console.log('Server running at http://localhost:8080');
});
```

4. アプリケーションを起動します。

```
$ node app.js
Server running at http://localhost:8080
```

5. **curl** またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で稼働していることを確認します。

```
$ curl http://localhost:8080
{"content":"Hello, World!"}
```

**追加情報**

- Node.js ランタイムは、Node.js API ドキュメント に記載されているコア Node.js API を提供します。

## 3.2. NODE.JS アプリケーションの OPENSHIFT へのデプロイ

Node.js アプリケーションを OpenShift にデプロイするには、**nodeshift** をアプリケーションに追加し、**package.json** ファイルを設定してから **nodeshift** を使用してデプロイします。

### 3.2.1. OpenShift デプロイメントに向けた Node.js アプリケーションの準備

OpenShift のデプロイメント用に Node.js アプリケーションを準備するには、以下の手順を実行する必要があります。

- **nodeshift** をアプリケーションに追加します。
- **openshift** および **start** エントリーを **package.json** ファイルに追加します。

**前提条件**

- **npm** がインストールされている。

**手順**

1. **nodeshift** をアプリケーションに追加します。

```
$ npm install nodeshift --save-dev
```

2. **openshift** および **start** エントリーを **package.json** の **scripts** セクションに追加します。

```
{
  "name": "myApp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "openshift": "nodeshift --expose --
dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12",
    "start": "node app.js",
    ...
```

```
  }
  ...
}
```

**openshift** スクリプトは **nodeshift** を使用して、アプリケーションを OpenShift にデプロイします。

**注記**

Universal Base Image および RHEL イメージが Node.js で利用できます。イメージ名の詳細は、Node.js リリースノートを参照してください。

3. **オプション**: **package.json** に **files** セクションを追加します。

```
{
  "name": "myApp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    ...
  },
  "files": [
    "package.json",
    "app.js"
  ]
  ...
}
```

**files** セクションは、**OpenShift** にデプロイするときに含めるファイルとディレクトリーを nodeshift に指示します。**nodeshift** は **node-tar** モジュールを使用して、**files** セクションにリストしたファイルとディレクトリーに基づいて tar ファイルを作成します。この tar ファイルは、**nodeshift** がアプリケーションを OpenShift にデプロイする際に使用されます。**files** セクションが指定されていない場合、**nodeshift** は現在のディレクトリー全体を送信します。ただし、以下は除外されます。

- **node_modules/**
- **.git/**
- **tmp/**
  OpenShift へのデプロイ時に不要なファイルを含めないように **package.json** に **files** セクションを含めることが推奨されます。

### 3.2.2. Node.js アプリケーションの OpenShift へのデプロイ

**nodeshift** を使用して Node.js アプリケーションを OpenShift にデプロイできます。

**前提条件**

- CLI クライアント **oc** がインストールされている。
- **npm** がインストールされている。

- ルートを設定する際に、アプリケーションが使用するポートがすべて正しく公開されていることを確認する。

**手順**

1. **oc** クライアントを使用して OpenShift インスタンスにログインします。

```
$ oc login ...
```

2. **nodeshift** を使用して、アプリケーションを OpenShift にデプロイします。

```
$ npm run openshift
```

## 3.3. NODE.JS アプリケーションのスタンドアロンの RED HAT ENTERPRISE LINUX へのデプロイ

**npm** を使用して、Node.js アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイできます。

**前提条件**

- Node.js アプリケーション。
- npm 6.14.8 がインストールされている。
- RHEL 7 または RHEL 8 がインストールされている。
- Node.js がインストールされている。

**手順**

1. プロジェクトの **package.json** ファイルに追加の依存関係を指定した場合は、アプリケーションを実行する前にその依存関係をインストールしてください。

```
$ npm install
```

2. アプリケーションの root ディレクトリーからアプリケーションをデプロイします。

```
$ node app.js
Server running at http://localhost:8080
```

**検証手順**

1. **curl** またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で実行していることを確認します。

```
$ curl http://localhost:8080
```

# 第4章 NODE.JS ベースのアプリケーションのデバッグ

このセクションでは、Node.js ベースのアプリケーションのデバッグと、ローカルデプロイメントとリモートデプロイメントの両方でデバッグロギングを使用する方法を説明します。

## 4.1. リモートのデバッグ

アプリケーションをリモートでデバッグするには、デバッグモードで起動し、デバッガーを割り当てる必要があります。

### 4.1.1. アプリケーションのローカルでの起動およびネイティブデバッガーの割り当て

ネイティブデバッガーを使用すると、ビルトインデバッグクライアントを使用して Node.js ベースのアプリケーションをデバッグできます。

**前提条件**

- デバッグするアプリケーション。

**手順**

1. デバッガーを有効にしてアプリケーションを起動します。
   ネイティブデバッガーは自動的に割り当てられ、デバッグプロンプトを提供します。

   **デバッガーが有効になっているサンプルアプリケーション**

   ```
   $ node inspect app.js
   < Debugger listening on ws://127.0.0.1:9229/12345678-aaaa-bbbb-cccc-0123456789ab
   < For help see https://nodejs.org/en/docs/inspector
   < Debugger attached.
   ...
   debug>
   ```

   アプリケーションに別のエントリーポイントがある場合は、コマンドを変更してそのエントリーポイントを指定する必要があります。

   ```
   $ node inspect path/to/entrypoint
   ```

   たとえば、express generator を使用してアプリケーションを作成する場合、エントリーポイントはデフォルトで **./bin/www** に設定されます。REST API Level 0 example application などのサンプルの一部は、エントリーポイントとして **./bin/www** を使用します。

2. デバッガープロンプトを使用して デバッグコマンド を実行します。

### 4.1.2. アプリケーションをローカルに起動して V8 インスペクターをアタッチする

V8 インスペクターを使用すると、Chrome Debugging Protocol を使用する Chrome DevTools などの他のツールを使用して Node.js ベースのアプリケーションをデバッグできます。

**前提条件**

- デバッグするアプリケーション。

- Google Chrome ブラウザーで 提供されるような V8 インスペクターがインストールされている。

**手順**

1. V8 インスペクターの統合を有効にして アプリケーションを起動します。

```
$ node --inspect app.js
```

アプリケーションに別のエントリーポイントがある場合は、コマンドを変更してそのエントリーポイントを指定する必要があります。

```
$ node --inspect path/to/entrypoint
```

たとえば、express generator を使用してアプリケーションを作成する場合、エントリーポイントはデフォルトで **./bin/www** に設定されます。REST API Level 0 example application などのサンプルの一部は、エントリーポイントとして **./bin/www** を使用します。

2. V8 インスペクターをアタッチし、デバッグコマンドを実行します。
たとえば、Google Chrome を使用している場合は、以下のようになります。

   a. **chrome://inspect** に移動します。

   b. 以下の **リモートターゲット** からアプリケーションを選択します。

   c. これで、アプリケーションのソースを確認し、デバッグアクションを実行できるようになりました。

### 4.1.3. デバッグモードでの OpenShift でのアプリケーションの起動

OpenShift で Node.js ベースのアプリケーションをリモートでデバッグするには、コンテナー内で **NODE_ENV** 環境変数を **development** に設定し、リモートデバッガーからアプリケーションに接続できるようにポート転送を設定する必要があります。

**前提条件**

- アプリケーションが OpenShift で実行している。
- **oc** バイナリーがインストールされている。
- ターゲット OpenShift 環境で **oc port-forward** コマンドを実行できる。

**手順**

1. **oc** コマンドを使用して、利用可能なデプロイメント設定を一覧表示します。

```
$ oc get dc
```

2. アプリケーションのデプロイメント設定の **NODE_ENV** 環境変数を **development** に設定して、デバッグを有効にします。以下に例を示します。

```
$ oc set env dc/MY_APP_NAME NODE_ENV=development
```

3. 設定変更時に自動的に再デプロイするように設定されていない場合は、アプリケーションを再デプロイします。以下に例を示します。

```
$ oc rollout latest dc/MY_APP_NAME
```

4. ローカルマシンからアプリケーション Pod へのポート転送を設定します。

   a. 現在実行中の Pod を一覧表示し、アプリケーションが含まれる Pod を検索します。

```
$ oc get pod
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-3-1xrsp         0/1       Running     0          6s
...
```

   b. ポート転送を設定します。

```
$ oc port-forward MY_APP_NAME-3-1xrsp $LOCAL_PORT_NUMBER:5858
```

   ここで、**$LOCAL_PORT_NUMBER** はローカルマシンで選択した未使用のポート番号になります。リモートデバッガー設定のこの番号を覚えておいてください。

5. V8 インスペクターをアタッチし、デバッグコマンドを実行します。
   たとえば、Google Chrome を使用している場合は、以下のようになります。

   a. **chrome://inspect** に移動します。

   b. **Configure** をクリックします。

   c. **127.0.0.1:$LOCAL_PORT_NUMBER** を追加します。

   d. **Done** をクリックします。

   e. 以下の **リモートターゲット** からアプリケーションを選択します。

   f. これで、アプリケーションのソースを確認し、デバッグアクションを実行できるようになりました。

6. デバッグが完了したら、アプリケーション Pod の **NODE_ENV** 環境変数の設定を解除します。
   以下に例を示します。

```
$ oc set env dc/MY_APP_NAME NODE_ENV-
```

## 4.2. デバッグロギング

デバッグロギングは、デバッグ時に詳細な情報をアプリケーションログに追加する方法です。これにより、以下が可能になります。

- アプリケーションの通常の操作中のロギングの出力を最小限に抑えて、読みやすさを改善し、ディスク領域の使用量を削減します。
- 問題の解決時にアプリケーションの内部作業に関する詳細情報を表示します。

### 4.2.1. デバッグロギングの追加

この例では、デバッグパッケージ を使用しますが、デバッグロギングを処理できる その他のパッケージも利用可能です。

**前提条件**

- デバッグするアプリケーション。たとえば、以下のようになります。

**手順**

1. **debug** ロギング定義を追加します。

```
const debug = require('debug')('myexample');
```

2. デバッグステートメントを追加します。

```
app.use('/api/greeting', (request, response) => {
  const name = request.query ? request.query.name : undefined;
  //log name in debugging
  debug('name: '+name);
  response.send({content: `Hello, ${name || 'World'}`});
});
```

3. デバッグ モジュールを **package.json** に追加します。

```
...
"dependencies": {
    "debug": "^3.1.0"
  }
```

アプリケーションによっては、このモジュールはすでに含まれている場合があります。たとえば、express generator を使用してアプリケーションを作成する場合、**debug** モジュールはすでに **package.json** に追加されています。REST API Level 0 サンプル などの一部のサンプルアプリケーションには、**package.json** ファイルに **debug** モジュールがすでにあります。

4. アプリケーションの依存関係をインストールします。

```
$ npm install
```

### 4.2.2. localhost でのデバッグログへのアクセス

アプリケーションを起動し、デバッグロギングを有効にする場合は、**DEBUG** 環境変数を使用します。

**前提条件**

- デバッグロギングを使用するアプリケーション。

**手順**

1. アプリケーションを起動し、デバッグロギングを有効にする場合は、**DEBUG** 環境変数を設定します。

```
$ DEBUG=myexample npm start
```

**debug** モジュールでは、ワイルドカード を使用してデバッグメッセージをフィルターできます。これは **DEBUG** 環境変数を使用して設定されます。

2. アプリケーションをテストしてデバッグロギングを呼び出します。
たとえば、REST API Level 0 のサンプル のデバッグロギングが、**/api/greeting** メソッドの **name** 変数をログに記録するように設定されている場合は、以下のようになります。

```
$ curl http://localhost:8080/api/greeting?name=Sarah
```

3. アプリケーションログを表示して、デバッグメッセージを表示します。

```
myexample name: Sarah +3m
```

### 4.2.3. OpenShift での Node.js デバッグログへのアクセス

OpenShift のアプリケーション Pod で **DEBUG** 環境変数を使用して、デバッグロギングを有効にします。

**前提条件**

- デバッグロギングを使用するアプリケーション。
- CLI クライアント **oc** がインストールされている。

**手順**

1. **oc** CLI クライアントを使用して、OpenShift インスタンスにログインします。

```
$ oc login ...
```

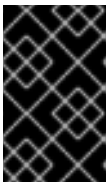
2. アプリケーションを OpenShift にデプロイします。

```
$ npm run openshift
```

これにより、**openshift** npm スクリプトを実行します。これは nodeshift への直接呼び出しをラップします。

3. Pod の名前を見つけ、ログを追跡して起動を監視します。

```
$ oc get pods
....
$ oc logs -f pod/POD_NAME
```



**重要**

Pod が起動したら、このコマンドを実行したままにして、新しいのターミナルウィンドウで残りの手順を実行します。これにより、ログを **追跡** でき、そのログの新しいエントリーを確認することができます。

4. アプリケーションをテストします。
たとえば、REST API Level 0 のサンプル にデバッグロギングがあり、**/api/greeting** メソッドの **name** 変数をログに記録した場合は、以下のようになります。

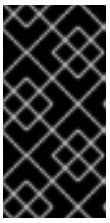
```
$ oc get routes
...
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

5. Pod ログに戻り、ログにデバッグロギングメッセージがないことに注意してください。
6. **DEBUG** 環境変数を設定して、デバッグロギングを有効にします。

```
$ oc get dc
...
$ oc set env dc DC_NAME DEBUG=myexample
```

7. Pod ログに戻り、更新ロールアウトを監視します。
   更新がロールアウトされると Pod が停止し、ログをフォローしなくなります。
8. 新規 Pod の名前を見つけ、ログを追跡します。

```
$ oc get pods
....
$ oc logs -f pod/POD_NAME
```



**重要**

Pod が起動したら、このコマンドを実行したままにして、別のターミナルウィンドウで残りの手順を実行します。これにより、ログを **追跡** でき、そのログの新しいエントリーを確認することができます。具体的には、ログにはデバッグメッセージが表示されます。

9. アプリケーションをテストして、デバッグロギングを呼び出します。

```
$ oc get routes
...
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

10. Pod ログに戻り、デバッグメッセージを表示します。

```
...
myexample name: Sarah +3m
```

デバッグロギングを無効にするには、Pod から **DEBUG** 環境変数を削除します。

```
$ oc set env dc DC_NAME DEBUG-
```

**関連情報**

環境変数についての詳細は、OpenShift のドキュメント を参照してください。

# 第5章 NODE.JS のサンプルアプリケーション

Node.js ランタイムは、サンプルアプリケーションを提供します。OpenShift でアプリケーションの開発を開始すると、サンプルアプリケーションをテンプレートとして使用できます。

これらのサンプルアプリケーションは Developer Launcher でアクセスできます。

すべてのサンプルアプリケーションを以下にダウンロードおよびデプロイできます。

- x86_64 アーキテクチャー - 本ガイドのサンプルアプリケーションでは、サンプルアプリケーションを x86_64 アーキテクチャーにビルドおよびデプロイする方法を説明します。
- s390x アーキテクチャー - IBM Z インフラストラクチャーでプロビジョニングされた OpenShift 環境にサンプルアプリケーションをデプロイするには、コマンドで関連する IBM Z イメージ名を指定します。
- ppc64le アーキテクチャー - IBM Power System インフラストラクチャーでプロビジョニングされている OpenShift 環境でサンプルアプリケーションをデプロイするには、コマンドに関連する IBM Power System のイメージ名を指定します。
  サンプルアプリケーションの一部には、ワークフローを実証するために Red Hat Data Grid などの他の製品も必要になります。この場合は、これらの製品のイメージ名を、サンプルアプリケーションの YAML ファイルで関連する IBM Z または IBM Power System のイメージ名に変更する必要もあります。

## 5.1. NODE.JS の REST API LEVEL 0 サンプル

**重要**

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

### REST API Level 0 サンプルでできること

REST API Level 0 のサンプルでは、REST フレームワークを使用して、HTTP 経由でビジネスオペレーションをリモートプロシージャコールエンドポイントにマッピングする方法が示されています。これは、Richardson Maturity Model の Level 0 に対応します。REST およびその基礎となる原則を使用して HTTP エンドポイントを作成すると、API のプロトタイプをすばやく作成して柔軟に設計できます。

この例では、HTTP プロトコルを使用してリモートサービスと対話するためのメカニズムが導入されました。これにより、以下が可能になります。

- **api/greeting** エンドポイントで HTTP **GET** 要求を実行します。
- **Hello, World!** で設定されるペイロードを使用して JSON 形式でレスポンスを受け取ります。文字列。
- String 引数を渡し、**api/greeting** エンドポイントで HTTP **GET** 要求を実行します。これにより、クエリー文字列に **name** 要求パラメーターが使用されます。
- **Hello, $name!** のペイロードを含む JSON 形式の応答を受け取ります。**$name** は、要求に渡される **name** パラメーターの値に置き換えられます。

### 5.1.1. REST API Level 0 設計トレードオフ

表5.1 設計トレードオフ

| 利点 | 不利な点 |
| --- | --- |
| • サンプルアプリケーションでは、高速なプロトタイプを有効にします。<br>• API Design には柔軟性があります。<br>• HTTP エンドポイントにより、クライアントは言語に依存しません。 | • アプリケーションまたはサービスが成熟するにつれて、REST API Level 0 アプローチは適切に拡張できない可能性があります。クリーンな API 設計や、データベースの対話に関するユースケースをサポートしない場合があります。<br>  ○ 共有された変更可能な状態を含むすべての操作は、適切なバッキングデータストアと統合する必要があります。<br>  ○ この API 設計で処理されるすべての要求は、要求に対応するコンテナーにのみスコープが指定されます。これ以降の要求は、同じコンテナーで処理されない可能性があります。 |

### 5.1.2. REST API Level 0 サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で REST API Level 0 サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.1.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.1.2.2. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.1.2.3. CLI クライアント oc を使用した REST API Level 0 サンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                       READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa            1/1       Running     0          58s
MY_APP_NAME-s2i-1-build        0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.1.3. REST API Level 0 サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、REST API Level 0 サンプルアプリケーションを Minishift または CDK でローカルに実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.1.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

**前提条件**

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

**手順**

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

**Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

### 5.1.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

### 5.1.3.3. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して Minishift または CDK でアプリケーションのサンプルを使用するには、Minishift または CDK Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.1.3.4. CLI クライアント oc を使用した REST API Level 0 サンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は、「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

   これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa             1/1       Running     0          58s
MY_APP_NAME-s2i-1-build         0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.1.4. REST API Level 0 サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

**手順**

- 「REST API Level 0 サンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

### 5.1.5. Node.js の未変更の REST API Level 0 サンプルアプリケーションとの対話

この例では、GET 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。

**前提条件**

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

**手順**

1. **curl** を使用して、サンプルに対して **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World!"}
```

2. **curl** を使用して、例に対して URL パラメーター **name** を使用して **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting?name=Sarah
{"content":"Hello, Sarah!"}
```

**注記**

ブラウザーから、例で提供されているフォームを使用して、これらの同じ対話を実行することもできます。このフォームは、プロジェクト **http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** のルートにあります。

### 5.1.6. REST リソース

REST の背景および関連情報は、以下を参照してください。

- Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)
- Richardson Maturity Model
- Express Web Framework
- Spring Boot の REST API Level 0 の例
- Eclipse Vert.x の REST API Level 0
- Thorntail の REST API Level 0

## 5.2. NODE.JS の外部化設定の例

**重要**

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

外部化設定は、ConfigMap を使用して設定を外部化する基本的な例を提供します。**ConfigMap** は、コンテナーを OpenShift に依存しないようにする一方で、単純なキーと値のペアとして設定データを 1 つ以上の Linux コンテナーに挿入するために OpenShift で使用されるオブジェクトです。

この例では、以下の方法を示しています。

- **ConfigMap** をセットアップし、設定します。
- アプリケーション内で **ConfigMap** によって提供される設定を使用します。
- 実行中のアプリケーションの **ConfigMap** 設定に変更をデプロイします。

### 5.2.1. 外部化された設定設計パターン

可能な場合は、アプリケーション設定を外部化し、アプリケーションコードから分離します。これにより、異なる環境を通過する際にアプリケーション設定を変更できますが、コードは変更されません。設定の外部化により、機密情報や内部情報がコードベースやバージョン管理から除外されます。多くの言語およびアプリケーションサーバーは、アプリケーション設定の外部化をサポートする環境変数を提供します。

マイクロサービスアーキテクチャーおよび多言語 (polyglot) 環境は、アプリケーションの設定を管理する複雑な層を追加します。アプリケーションは独立した分散サービスで設定され、それぞれ独自の設定を持つことができます。すべての設定データを同期し、アクセス可能な状態に維持すると、メンテナーンスの課題が発生します。

ConfigMap により、アプリケーション設定を外部化でき、OpenShift 上の個別の Linux コンテナーおよび Pod で使用できます。YAML ファイルの使用を含むさまざまな方法で ConfigMap オブジェクトを作成し、これを Linux コンテナーに挿入できます。ConfigMap を使用すると、設定データの分類およびスケーリングが可能です。これにより、基本的な **Development**、**Stage**、および **Production** 以外の多くの環境を設定できます。ConfigMap の詳細は、OpenShift ドキュメント を参照してください。

### 5.2.2. 外部化設定設計のトレードオフ

**表5.2 設計のトレードオフ**

| 利点 | 不利な点 |
| --- | --- |
| • 設定がデプロイメントと分離している<br>• 個別に更新が可能<br>• サービス間で共有できる | • 環境への設定の追加には追加のステップが必要<br>• 個別に保守する必要がある<br>• サービスの範囲を超える調整が必要 |

### 5.2.3. 外部化設定サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で外部化設定アプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.2.3.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.2.3.2. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.2.3.3. CLI クライアント oc を使用した Externalized Configuration サンプルアプリケーションのデプロイメント

このセクションでは、外部化設定のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. サンプルアプリケーションをデプロイする前に、サービスアカウントに表示アクセス権を割り当て、ConfigMap の内容を読み取るためにアプリケーションが OpenShift API にアクセスできるようにします。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

4. アプリケーションの root ディレクトリーに移動します。

5. **app-config.yml** を使用して ConfigMap 設定を OpenShift にデプロイします。

```
$ oc create configmap app-config --from-file=app-config.yml
```

6. ConfigMap 設定がデプロイされていることを確認します。

```
$ oc get configmap app-config -o yaml

apiVersion: template.openshift.io/v1
data:
  app-config.yml: |-
    message : "Hello, %s from a ConfigMap !"
    level : INFO
...
```

7. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

8. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY    STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa           1/1      Running     0          58s
MY_APP_NAME-s2i-1-build       0/1      Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

9. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.2.4. 外部化設定サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK で外部設定サンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.2.4.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

**前提条件**

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

**手順**

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

**Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer
```

```
To login as administrator:
    oc login -u system:admin
```

#### 5.2.4.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.2.4.3. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して Minishift または CDK でアプリケーションのサンプルを使用するには、Minishift または CDK Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.2.4.4. CLI クライアント oc を使用した Externalized Configuration サンプルアプリケーションのデプロイメント

このセクションでは、外部化設定のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は、「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. サンプルアプリケーションをデプロイする前に、サービスアカウントに表示アクセス権を割り当て、ConfigMap の内容を読み取るためにアプリケーションが OpenShift API にアクセスできるようにします。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

4. アプリケーションの root ディレクトリーに移動します。
5. **app-config.yml** を使用して ConfigMap 設定を OpenShift にデプロイします。

```
$ oc create configmap app-config --from-file=app-config.yml
```

6. ConfigMap 設定がデプロイされていることを確認します。

```
$ oc get configmap app-config -o yaml

apiVersion: template.openshift.io/v1
data:
  app-config.yml: |-
      message : "Hello, %s from a ConfigMap !"
      level : INFO
...
```

7. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

   これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

8. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                           READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa            1/1       Running     0          58s
MY_APP_NAME-s2i-1-build        0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

9. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.2.5. 外部設定サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

**手順**

- 「外部化設定サンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

### 5.2.6. Node.js の未変更の外部化設定サンプルアプリケーションとの対話

この例では、**GET** 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。

**前提条件**

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

**手順**

1. **curl** を使用して、サンプルに対して **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World from a ConfigMap !"}
```

2. デプロイされた ConfigMap 設定を更新します。

```
$ oc edit configmap app-config
```

**message** キーの値を **Bonjour, %s from a ConfigMap !** に変更し、ファイルを保存します。

3. ConfigMap の更新は、アプリケーションの再起動を必要とせずに許容可能な時間 (数秒) 内でアプリケーションによって読み取る必要があります。

4. 更新された ConfigMap 設定を使用した例に対して **curl** を使用して **GET** 要求を実行し、更新されたグリーティングを確認します。また、アプリケーションによって提供される Web フォームを使用して、ブラウザーから実行することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Bonjour, World from a ConfigMap !"}
```

### 5.2.7. 外部化設定リソース

外部化設定および ConfigMap の背景および関連情報は、以下を参照してください。

- OpenShift ConfigMap ドキュメント
- OpenShift での ConfigMap に関するブログ投稿
- Spring Boot の外部化設定の例
- Eclipse Vert.x の外部化設定
- Thorntail の Externalized Configuration

## 5.3. NODE.JS の RELATIONAL DATABASE BACKEND のサンプル

**重要**

以下の例は、実稼働環境での実行を目的としていません。

**制限:** このサンプルアプリケーションを Minishift または CDK で実行します。手動ワークフローを使用して、このサンプルを OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Foundational**

#### リレーショナルデータベースバックエンドのサンプル

Relational Database Backend のサンプルは、REST API Level 0 アプリケーションを拡張し、単純な HTTP API を使用して PostgreSQL データベースで **作成**、**読み取り**、**更新**、および **削除** (**CRUD**) 操作を実行する基本的な例を提供します。**CRUD** 操作は永続ストレージの 4 つの基本的な機能であり、データベースを処理する HTTP API の開発時に広く使用されます。

また、この例では、HTTP アプリケーションが OpenShift のデータベースを見つけ、接続する機能も示しています。各ランタイムは、指定したケースで最も適した接続ソリューションを実装する方法を示しています。ランタイムは、**JDBC**、**JPA**の使用、または **ORM** API に直接アクセスするなどのオプションを選択できます。

サンプルアプリケーションは HTTP API を公開し、HTTP で **CRUD** 操作を実行してデータを操作できるようにするエンドポイントを提供します。**CRUD** 操作は HTTP **Verbs** にマップされます。API は JSON フォーマットを使用して要求を受け取り、ユーザーに応答を返します。また、ユーザーは、サンプルが提供するユーザーインターフェイスを使用して、アプリケーションを使用することもできます。具体的には、この例では以下を可能にするアプリケーションを提供します。

- ブラウザーでアプリケーション Web インターフェイスに移動します。これにより、**my_data** データベースのデータで **CRUD** 操作を実行する簡単な Web サイトが公開されます。
- **api/fruits** エンドポイントで HTTP **GET** 要求を実行します。
- データベース内のすべての fruits の一覧が含まれる JSON 配列としてフォーマットされたレスポンスを受け取ります。
- 有効なアイテム ID を引数として渡しながら、**api/fruits/*** エンドポイントで HTTP **GET** 要求を実行します。
- 指定の ID を持つ fruit の名前が含まれる JSON 形式で応答を受け取ります。指定された ID に項目がない場合は、呼び出しにより HTTP エラー 404 が発生します。
- **api/fruits** エンドポイントで HTTP **POST** 要求を実行し、有効な **name** 値を渡してデータベースの新規エントリーを作成します。
- 有効な ID および名前を引数として渡す **api/fruits/*** エンドポイントで HTTP **PUT** 要求を実行します。これにより、要求に指定された名前に一致するように、指定の ID を持つ項目の名前が更新されます。
- **api/fruits/*** エンドポイントで HTTP **DELETE** 要求を実行し、有効な ID を引数として渡します。これにより、指定された ID の項目がデータベースから削除され、応答として HTTP コード **204** (コンテンツなし) を返します。無効な ID を渡すと、呼び出しにより HTTP エラー **404** が発生します。

この例では、完全に成熟した RESTful モデル (レベル 3) を示していませんが、推奨される HTTP API プラクティスに従って、互換性のある HTTP 動詞およびステータスを使用しています。

### 5.3.1. Relational Database Backend の設計トレードオフ

**表5.3 設計のトレードオフ**

| 利点 | 不利な点 |
|---|---|

| 利点 | 不利な点 |
| --- | --- |
| • 各ランタイムは、データベースの対話の実装方法を決定します。もう 1 つは JDBC などの低レベルの接続 API を使用すると、JPA を使用できますが、もう 1 つは ORM API に直接アクセスできます。各ランタイムは、最適な方法を決定します。<br>• 各ランタイムはスキーマの作成方法を決定します。 | • このサンプルアプリケーションで提供される PostgreSQL データベースは永続ストレージではバックアップされていません。データベース Pod を停止または再デプロイすると、データベースへの変更は失われます。変更を保持するために、サンプルアプリケーションの Pod で外部データベースを使用するには、OpenShift ドキュメントの Creating an application with a database を参照してください。OpenShift 上のデータベースコンテナーで永続ストレージを設定することもできます。OpenShift およびコンテナーで永続ストレージを使用する方法は、OpenShift ドキュメントの Persistent Storage、Managing Volumes および Persistent Volumes の章を参照してください。 |

### 5.3.2. Relational Database Backend のサンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で Relational Database Backend アプリケーションのサンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.3.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.3.2.2. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.3.2.3. CLI クライアント oc を使用した Relational Database Backend サンプルアプリケーションのデプロイメント

このセクションでは、Relational Database Backend サンプルアプリケーションを構築し、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. PostgreSQL データベースを OpenShift にデプロイします。データベースアプリケーションの作成時に、ユーザー名、パスワード、データベース名に以下の値を使用するようにしてください。これらの値を使用するために、サンプルアプリケーションが事前設定されています。異なる値を使用すると、アプリケーションがデータベースと統合できなくなります。

```
$ oc new-app -e POSTGRESQL_USER=luke -ePOSTGRESQL_PASSWORD=secret -
ePOSTGRESQL_DATABASE=my_data registry.access.redhat.com/rhscl/postgresql-10-rhel7
--name=my-database
```

5. データベースのステータスを確認し、Pod が実行されていることを確認します。

```
$ oc get pods -w
my-database-1-aaaaa   1/1       Running   0         45s
my-database-1-deploy   0/1       Completed   0         53s
```

**my-database-1-aaaaa** Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
MY_APP_NAME-s2i-1-build   0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。

8. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                PATH      SERVICES          PORT
TERMINATION
MY_APP_NAME   MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME   8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.3.3. Relational Database Backend サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK でローカルで Relational Database Backend サンプルアプリケーションを実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.3.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

**前提条件**

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

**手順**

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

**Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

#### 5.3.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.3.3.3. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して Minishift または CDK でアプリケーションのサンプルを使用するには、Minishift または CDK Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.3.3.4. CLI クライアント oc を使用した Relational Database Backend サンプルアプリケーションのデプロイメント

このセクションでは、Relational Database Backend サンプルアプリケーションを構築し、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は、「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

- 

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. PostgreSQL データベースを OpenShift にデプロイします。データベースアプリケーションの作成時に、ユーザー名、パスワード、データベース名に以下の値を使用するようにしてください。これらの値を使用するために、サンプルアプリケーションが事前設定されています。異なる値を使用すると、アプリケーションがデータベースと統合できなくなります。

```
$ oc new-app -e POSTGRESQL_USER=luke -ePOSTGRESQL_PASSWORD=secret -
ePOSTGRESQL_DATABASE=my_data registry.access.redhat.com/rhscl/postgresql-10-rhel7
--name=my-database
```

5. データベースのステータスを確認し、Pod が実行されていることを確認します。

```
$ oc get pods -w
my-database-1-aaaaa   1/1       Running   0         45s
my-database-1-deploy   0/1       Completed   0         53s
```

**my-database-1-aaaaa** Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
MY_APP_NAME-s2i-1-build   0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。

8. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                PATH      SERVICES          PORT
TERMINATION
```

```
MY_APP_NAME   MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME   8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.3.4. Relational Database Backend サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

**手順**

- 「Relational Database Backend のサンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

### 5.3.5. Node.js での Relational Database Backend API との対話

サンプルアプリケーションの作成が完了したら、以下のように対話できます。

**前提条件**

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

**手順**

1. 以下のコマンドを実行して、アプリケーションの URL を取得します。

```
$ oc get route MY_APP_NAME
```

```
NAME            HOST/PORT                                   PATH    SERVICES        PORT
TERMINATION
MY_APP_NAME     MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME     8080
```

2. データベースアプリケーションの Web インターフェイスにアクセスするには、ブラウザーで **アプリケーション URL** に移動します。

```
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
```

または、**curl** を使用して **api/fruits/*** エンドポイントで要求を直接作成できます。

**データベースのエントリーの一覧を表示します。**

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits
```

```
[ {
  "id" : 1,
  "name" : "Apple",
  "stock" : 10
}, {
  "id" : 2,
  "name" : "Orange",
  "stock" : 10
}, {
  "id" : 3,
  "name" : "Pear",
  "stock" : 10
} ]
```

**特定の ID のエントリーの取得**

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/3
```

```
{
  "id" : 3,
  "name" : "Pear",
  "stock" : 10
}
```

**エントリーを新規作成します。**

```
$ curl -H "Content-Type: application/json" -X POST -d '{"name":"Peach","stock":1}'
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits
```

```
{
  "id" : 4,
  "name" : "Peach",
  "stock" : 1
}
```

**エントリーを更新します。**

```
$ curl -H "Content-Type: application/json" -X PUT -d '{"name":"Apple","stock":100}'
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/1
```

```
{
  "id" : 1,
  "name" : "Apple",
  "stock" : 100
}
```

**エントリーを削除します。**

```
$ curl -X DELETE http://MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/1
```

**トラブルシューティング**

- これらのコマンドを実行後に HTTP エラーコード **503** を応答として受け取った場合は、アプリケーションが準備状態にないことを意味します。

### 5.3.6. リレーショナルデータベースリソース

OpenShift、CRUD、HTTP API、および REST でのリレーショナルデータベースの実行に関する背景および関連情報は、以下を参照してください。

- HTTP Verbs
- Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)
- The never ending REST API design debase
- REST APIs must be Hypertext driven
- Richardson Maturity Model
- Express Web Framework
- Relational Database Backend for Spring Boot
- Relational Database Backend for Eclipse Vert.x
- Relational Database Backend for Thorntail

## 5.4. NODE.JS のヘルスチェックの例

**重要**

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

アプリケーションをデプロイする場合は、アプリケーションが利用可能かどうかを確認し、受信した要求の処理を開始することが重要です。**ヘルスチェック** パターンを実装すると、アプリケーションが利用できるかどうかや要求に対応できるかどうかなど、アプリケーションの健全性を監視できます。

**注記**

ヘルスチェックの用語に慣れていない場合は、最初に 「ヘルスチェックの概念」 セクションを参照してください。

このユースケースの目的は、プローブを使用してヘルスチェックパターンを実証することです。プロービングは、アプリケーションの liveness および readiness を報告するために使用されます。このユースケースでは、HTTP **health** エンドポイントを公開して HTTP 要求を発行するアプリケーションを設定します。コンテナーが動作している場合は、HTTP エンドポイント **health** の Liveness プローブによると、管理プラットフォームは **200** を戻りコードとして受け取り、それ以上のアクションは必要ありませ

ん。HTTP エンドポイント **health** が応答を返さない場合 (たとえば、スレッドがブロックされている場合)、アプリケーションは liveness プローブにより有効とみなされません。この場合、プラットフォームはそのアプリケーションに対応する Pod を強制終了し、アプリケーションを再起動するために新規 Pod を再作成します。

このユースケースでは、readiness プローブを実証し、使用することもできます。アプリケーションが実行していても要求を処理できない場合 (再起動中にアプリケーションが HTTP **503** 応答コードを返す場合など)、このアプリケーションは readiness プローブにより準備ができていないと見なされます。readiness プローブによってアプリケーションが準備状態にあるとみなされない場合、要求は readiness プローブにより準備状態にあるとみなされるまで、要求はそのアプリケーションにルーティングされません。

### 5.4.1. ヘルスチェックの概念

ヘルスチェックパターンを理解するには、まず以下の概念を理解する必要があります。

**Liveness**

liveness は、アプリケーションが実行しているかどうかを定義します。実行中のアプリケーションが応答しない状態または停止状態に移行する可能性があるため、再起動する必要があります。liveness の確認は、アプリケーションを再起動する必要があるかどうかを判断するのに役立ちます。

**Readines**

readiness は、実行中のアプリケーションが要求を処理できるかどうかを定義します。実行中のアプリケーションがエラーまたは破損状態に切り替わり、要求にサービスを提供することができません。readiness を確認すると、要求が引き続きそのアプリケーションにルーティングされるべきかどうかが判断されます。

**フェイルオーバー**

フェイルオーバーにより、サービス要求の失敗が適切に処理されるようになります。アプリケーションが要求のサービスに失敗した場合、その要求と今後の要求は **フェイルオーバー** したり、別のアプリケーションにルーティングしたりすることができます。通常、同じアプリケーションの冗長コピーになります。

**耐障害性および安定性**

耐障害性および安定性により、要求サービスの失敗を適切に処理できます。接続の損失によりアプリケーションが要求を処理できない場合、耐障害性のあるシステムでは、接続が再確立された後に要求を再試行できます。

**プローブ**

プローブは実行中のコンテナーで定期的に実行する Kubernetes の動作です。

### 5.4.2. Health Check サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で Health Check サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.4.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.4.2.2. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.4.2.3. CLI クライアント oc を使用した Health Check サンプルアプリケーションのデプロイメント

このセクションでは、Health Check のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                       READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa            1/1       Running     0          58s
MY_APP_NAME-s2i-1-build        0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.4.3. Health Check サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK で Health Check サンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.4.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

**前提条件**

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

**手順**

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

**Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

#### 5.4.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.4.3.3. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して Minishift または CDK でアプリケーションのサンプルを使用するには、Minishift または CDK Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.4.3.4. CLI クライアント oc を使用した Health Check サンプルアプリケーションのデプロイメント

このセクションでは、Health Check のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は、「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

- 

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. **npm** を使用して OpenShift へのデプロイメントを開始します。

```
$ npm install && npm run openshift
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                     READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa      1/1       Running     0          58s
MY_APP_NAME-s2i-1-build  0/1       Completed   0          2m
```

**MY_APP_NAME-1-aaaaa** Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.4.4. Health Check サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

**手順**

- 「Health Check サンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

### 5.4.5. 未変更の Health Check サンプルアプリケーションとの対話

サンプルアプリケーションをデプロイすると、**MY_APP_NAME** サービスが実行します。**MY_APP_NAME** サービスは、以下の REST エンドポイントを公開します。

**/api/greeting**

**name** パラメーター (または World をデフォルト値として) のグリーティングが含まれる JSON を返します。

**/api/stop**

障害をシミュレートする手段として、サービスが強制的に応答しなくなります。

以下の手順は、サービスの可用性を確認し、障害をシミュレートする方法を示しています。この利用可能なサービスの障害により、OpenShift の自己修復機能がサービスでトリガーされます。

Web インターフェイスを使用して、これらの手順を実施することもできます。

1. **curl** を使用して、**MY_APP_NAME** サービスに対して **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
```

```
{"content":"Hello, World!"}
```

2. **/api/stop** エンドポイントを呼び出して、その直後に **/api/greeting** エンドポイントの可用性を確認します。
**/api/stop** エンドポイントを呼び出すと、内部サービス障害をシミュレートし、OpenShift の自己修復機能をトリガーします。障害のシミュレート後に **/api/greeting** を呼び出すと、サービスは HTTP ステータス **503** を返すはずです。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/stop
```

```
Stopping HTTP server, Bye bye world !
```

(続く)

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
```

```
Not online
```

3. **oc get pods -w** を使用して、動作中の自己修復機能を継続的に監視します。
サービス障害の呼び出し中に、OpenShift コンソールまたは **oc** クライアントツールで機能する自己修復機能を確認できます。**READY** 状態の Pod 数がゼロ (**0/1**) になり、短期間 (1 分未満) 後に 1 (**1/1**) に戻ることが確認できるはずです。さらに、サービスの失敗を呼び出すたびに **RESTARTS** カウントが増加します。

```
$ oc get pods -w
NAME                    READY    STATUS    RESTARTS   AGE
MY_APP_NAME-1-26iy7   0/1      Running   5          18m
MY_APP_NAME-1-26iy7   1/1      Running   5          19m
```

4. 必要に応じて、Web インターフェイスを使用してサービスを呼び出します。
端末ウィンドウを使用した対話では、サービスが提供する Web インターフェイスを使用して、異なるメソッドを起動し、サービスがライフサイクルフェーズを進めるのを監視できます。

```
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
```

5. 必要に応じて、Web コンソールを使用して、自己修復プロセスの各段階でアプリケーションによって生成されたログ出力を表示します。

    1. プロジェクトに移動します。
    2. サイドバーで **Monitoring** をクリックします。
    3. 画面右上にある **Events** をクリックし、ログメッセージを表示します。
    4. オプション: **View Details** をクリックし、Event ログの詳細なビューを表示します。

    ヘルスチェックアプリケーションは以下のメッセージを生成します。

| メッセージ | Status |
|---|---|
| **Unhealthy** | readiness プローブが失敗しました。このメッセージは想定されており、**/api/greeting** エンドポイントのシミュレートされた失敗が検出され、自己修復プロセスが開始することを示します。 |
| **Killing** | サービスを実行している利用不可の Docker コンテナーは、再作成前に強制終了されています。 |
| **Pulling** | 最新バージョンの Docker イメージをダウンロードして、コンテナーを再作成します。 |
| **Pulled** | Docker イメージが正常にダウンロードされました。 |
| **Created** | Docker コンテナーが正常に作成されました |
| **Started** | Docker コンテナーが要求を処理する準備ができました。 |

### 5.4.6. ヘルスチェックのリソース

ヘルスチェックの背景および関連情報は、以下を参照してください。

- OpenShift のアプリケーションの正常性

- Kubernetes Liveness and Readiness Probes
- Spring Boot のヘルスチェックの例
- Eclipse Vert.x のヘルスチェック
- Thorntail のヘルスチェック

## 5.5. NODE.JS のサーキットブレーカーの例

**重要**

以下の例は、実稼働環境での実行を目的としていません。

**制限:** このサンプルアプリケーションを Minishift または CDK で実行します。手動ワークフローを使用して、このサンプルを OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Foundational**

**Circuit Breaker** の例は、サービスの障害を報告し、要求の処理に使用できるまで失敗したサービスへのアクセスを制限する一般的なパターンを示しています。これにより、障害が発生したサービスの機能に依存する他のサービスでのカスケード障害を防ぐことができます。

この例では、サービスに Circuit Breaker および Fallback パターンを実装する方法を示しています。

### 5.5.1. Circuit Breaker 設計パターン

Circuit Breaker は、以下を行うパターンです。

- ネットワーク障害の影響を減らし、サービスが他のサービスを同期的に呼び出すサービスアーキテクチャーのレイテンシーを高くします。
  サービスの 1 つが以下である場合:
  - ネットワーク障害により利用できない
  - 負荷がかかっているトラフィックが原因で、レイテンシーが異常に高くなる

  エンドポイントを呼び出そうとする他のサービスは、エンドポイントに到達しようとして重要なリソースを使い果たし、使用できなくなる可能性があります。
- マイクロサービスアーキテクチャー全体が使用できなくなる可能性があるカスケード障害とも呼ばれる状態を防止します。
- 障害を監視する保護機能とリモート機能の間のプロキシーとして機能します。
- 障害が特定のしきい値に達するとトリップし、サーキットブレーカーへのそれ以降のすべての呼び出しは、保護された呼び出しがまったく行われずに、エラーまたは事前定義されたフォールバック応答を返します。

Circuit Breaker には通常、Circuit Breaker が作動した時に通知するエラー報告メカニズムも含まれています。

**Circuit Breaker の実装**

- Circuit Breaker パターンが実装されると、サービスクライアントは一定間隔でプロキシー経由でリモートサービスエンドポイントを呼び出します。
- リモートサービスエンドポイントへの呼び出しが繰り返し一貫して失敗すると、Circuit Breaker が作動し、サービスへのすべての呼び出しが、設定されたタイムアウト期間に即座に失敗し、事前定義されたフォールバック応答を返します。
- タイムアウト期間が期限切れになると、限られた数のテストコールがリモートサービスを通過して、それを修復したかどうかを判断したり、利用不可のままであるかを判断できます。
  - テスト呼び出しが失敗すると、Circuit Breaker はサービスを利用できない状態に維持し、受信呼び出しへのフォールバック応答を返します。
  - テストに成功すると、Circuit Breaker が閉じ、トラフィックが再度リモートサービスに到達できるようにします。

### 5.5.2. Circuit Breaker 設計のトレードオフ

**表5.4 設計のトレードオフ**

| 利点 | 不利な点 |
| --- | --- |
| • サービスが、呼び出す他のサービスの失敗を処理できるようにします。 | • タイムアウト値の最適化が困難な場合がある<br>  ◦ タイムアウト値を必要以上に大きくすると、レイテンシーが過度に生成される可能性があります。<br>  ◦ タイムアウト値を必要な値より小さくすると、誤検出が発生する可能性があります。 |

### 5.5.3. Red Hat build of Node.js のサーキットブレーカーアドオン

Red Hat は、Red Hat build of Node.js 向けのサーキットブレーカーアドオンを提供します。これは、オープンソースの Opossum モジュールの完全にサポートされた実装です。Red Hat カスタマーレジストリーから、サーキットブレーカーアドオンをダウンロードしてインストールできます。Node.js アプリケーションでサーキットブレーカーアドオンを使用して、アプリケーションの障害を監視し、適切に処理することができます。サーキットブレーカーアドオンを使用する場合は、アプリケーション障害が発生するたびに、指定したアクションを実行するフォールバック関数を定義することもできます。イベントハンドラーを定義して、さまざまなタイプのサーキットブレーカーイベントをリッスンすることもできます。

**関連情報**

- サーキットブレーカーアドオンの使用に関する詳細は、Red Hat build of Node.js 向けのサーキットブレーカーアドオンのスタートガイド を参照してください。

### 5.5.4. Circuit Breaker サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で Circuit Breaker サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.5.4.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.5.4.2. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.5.4.3. CLI クライアント oc を使用した Circuit Breaker サンプルアプリケーションのデプロイメント

このセクションでは、Circuit Breaker サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. 提供される **start-openshift.sh** スクリプトを使用して、OpenShift へのデプロイメントを開始します。

```
$ chmod +x start-openshift.sh
$ ./start-openshift.sh
```

   これらのコマンドは Nodeshift **npm** モジュールを使用して依存関係をインストールし、OpenShift で S2I ビルドプロセスを起動して、サービスを起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
MY_APP_NAME-greeting-1-aaaaa     1/1       Running   0          17s
MY_APP_NAME-greeting-1-deploy    0/1       Completed 0          22s
MY_APP_NAME-name-1-aaaaa         1/1       Running   0          14s
MY_APP_NAME-name-1-deploy        0/1       Completed 0          28s
```

   完全にデプロイされて起動すると、**MY_APP_NAME-greeting-1-aaaaa** Pod と **MY_APP_NAME-name-1-aaaaa** Pod の両方のステータスが **Running** となります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-greeting-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-greeting   8080
None
MY_APP_NAME-name       MY_APP_NAME-name-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-name       8080
None
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.5.5. Circuit Breaker サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のいずれかのオプションを使用して、Minishift または CDK で Circuit Breaker サンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.5.5.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

**前提条件**

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

**手順**

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

**Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443
```

```
You are logged in as:
    User:     developer
    Password: developer

To login as administrator:
    oc login -u system:admin
```

#### 5.5.5.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.5.5.3. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して Minishift または CDK でアプリケーションのサンプルを使用するには、Minishift または CDK Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.5.5.4. CLI クライアント oc を使用した Circuit Breaker サンプルアプリケーションのデプロイメント

このセクションでは、Circuit Breaker サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は、「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. 提供される **start-openshift.sh** スクリプトを使用して、OpenShift へのデプロイメントを開始します。

```
$ chmod +x start-openshift.sh
$ ./start-openshift.sh
```

   これらのコマンドは Nodeshift **npm** モジュールを使用して依存関係をインストールし、OpenShift で S2I ビルドプロセスを起動して、サービスを起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
MY_APP_NAME-greeting-1-aaaaa     1/1       Running   0          17s
MY_APP_NAME-greeting-1-deploy    0/1       Completed 0          22s
MY_APP_NAME-name-1-aaaaa         1/1       Running   0          14s
MY_APP_NAME-name-1-deploy        0/1       Completed 0          28s
```

   完全にデプロイされて起動すると、**MY_APP_NAME-greeting-1-aaaaa** Pod と **MY_APP_NAME-name-1-aaaaa** Pod の両方のステータスが **Running** となります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されま

す。たとえば、**MY_APP_NAME-greeting-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新しいビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME                HOST/PORT                                   PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-greeting   8080
None
MY_APP_NAME-name       MY_APP_NAME-name-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-name       8080
None
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

### 5.5.6. Circuit Breaker サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

**手順**

- 「Circuit Breaker サンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

### 5.5.7. 未変更の Node.js サーキットブレーカーサンプルアプリケーションとの対話

Node.js のサンプルアプリケーションをデプロイした後に、以下のサービスが実行されます。

**MY_APP_NAME-name**

以下のエンドポイントを公開します。

- このサービスの稼働時に名前を返す **/api/name** エンドポイントと、このサービスが失敗を実証するように設定されたときにエラーが返されます。
- **/api/state** エンドポイント (**/api/name** エンドポイントの動作を制御し、サービスが正しく機能するか、または失敗を示すかどうかを判断する)

**MY_APP_NAME-greeting**

以下のエンドポイントを公開します。

- パーソナライズされたグリーティング応答を取得するために呼び出す **/api/greeting** エンドポイント。
  **/api/greeting** エンドポイントを呼び出すと、要求の処理の一環として、**MY_APP_NAME-name** サービスの **/api/name** エンドポイントに対して呼び出しを実行します。**/api/name** エンドポイントに対する呼び出しは、Circuit Breaker によって保護されます。

  リモートエンドポイントが利用可能な場合、**name** サービスは HTTP コード **200** (**OK**) で応答し、**/api/greeting** エンドポイントから以下のグリーティングを受け取ります。

  ```
  {"content":"Hello, World!"}
  ```

  リモートエンドポイントが利用できない場合、**name** サービスは HTTP コード **500** (**Internal server error**) で応答し、**/api/greeting** エンドポイントから事前定義されたフォールバック応答を受け取ります。

  ```
  {"content":"Hello, Fallback!"}
  ```

- Circuit Breaker の状態を返す **/api/cb-state** エンドポイント。状態は以下のとおりです。
  - **open** : サーキットブレーカーにより、失敗したサービスに要求が到達できないようになります。
  - **closed**: サーキットブレーカーにより、要求がサービスに到達できるようになります。

以下の手順は、サービスの可用性を確認し、障害をシミュレートしてフォールバック応答を受け取る方法を示しています。

1. **curl** を使用して、**MY_APP_NAME-greeting** サービスに対して **GET** 要求を実行します。これを行うには、Web インターフェイスの **Invoke** ボタンを使用することもできます。

   ```
   $ curl http://MY_APP_NAME-greeting-
   MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
   {"content":"Hello, World!"}
   ```

2. **MY_APP_NAME-name** サービスの失敗をシミュレートするには、以下を行います。
   - Web インターフェイスで **Toggle** ボタンを使用します。
   - **MY_APP_NAME-name** サービスを実行している Pod のレプリカ数を 0 にスケーリングします。
   - **MY_APP_NAME-name** サービスの **/api/state** エンドポイントに対して HTTP **PUT** 要求を実行し、その状態を **fail** に設定します。

     ```
     $ curl -X PUT -H "Content-Type: application/json" -d '{"state": "fail"}'
     http://MY_APP_NAME-name-
     MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/state
     ```

3. **/api/greeting** エンドポイントを呼び出します。**/api/name** エンドポイントで複数の要求が失敗する場合:
   a. Circuit Breaker が開きます。
   b. Web インターフェイスの状態インジケーターが **CLOSED** から **OPEN** に変わります。

c. **/api/greeting** エンドポイントを呼び出すと、Circuit Breaker はフォールバック応答を実行します。

```
$ curl http://MY_APP_NAME-greeting-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, Fallback!"}
```

4. 名前 **MY_APP_NAME-name** サービスを可用性に戻します。これを実行するには、以下を行います。

- Web インターフェイスで **Toggle** ボタンを使用します。
- **MY_APP_NAME-name** サービスを実行している Pod のレプリカ数を 1 にスケーリングします。
- **MY_APP_NAME-name** サービスの **/api/state** エンドポイントに対して HTTP **PUT** 要求を実行し、その状態を **ok** に戻します。

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"state": "ok"}'
http://MY_APP_NAME-name-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/state
```

5. **/api/greeting** エンドポイントを再度呼び出します。**/api/name** エンドポイントでの複数の要求が正常に実行する場合:

a. Circuit Breaker が閉じます。

b. Web インターフェイスの状態インジケーターが **OPEN** から **CLOSED** に変わります。

c. Circuit Breaker は、**/api/greeting** エンドポイントを呼び出す際に **Hello World!** グリーティングを返します。

```
$ curl http://MY_APP_NAME-greeting-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World!"}
```

### 5.5.8. Circuit Breaker リソース

Circuit Breaker パターンの背後にある設計原則に関する背景情報については、以下のリンクを参照してください。

- Red Hat build of Node.js 向けのサーキットブレーカーアドオンのスタートガイド
- microservices.io: Microservice Patterns: Circuit Breaker
- Martin Fowler: CircuitBreaker
- Spring Boot のサーキットブレーカーの例
- Eclipse Vert.x の Circuit Breaker
- Thorntail の Circuit Breaker

## 5.6. NODE.JS のセキュアなサンプルアプリケーション

**重要**

以下の例は、実稼働環境での実行を目的としていません。

**制限:** このサンプルアプリケーションを Minishift または CDK で実行します。手動ワークフローを使用して、このサンプルを OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Advanced**

Secured サンプルアプリケーションは、Red Hat SSO を使用して REST エンドポイントを保護します。この例では、REST API Level 0 の例に展開されます。

Red Hat SSO

- OAuth 2.0 仕様のエクステンションである Open ID Connect プロトコルを実装します。
- アクセストークンを発行し、セキュアなリソースに対するさまざまなアクセス権限をクライアントに提供します。

SSO でアプリケーションをセキュアにすると、セキュリティー設定を一元化しつつ、アプリケーションにセキュリティーを追加することができます。

**重要**

この例にはデモ目的で Red Hat SSO が事前設定されており、その原則、使用方法、または設定を説明しません。この例を使用する前に、Red Hat SSO に関する基本的な概念を理解していることを確認してください。

### 5.6.1. Secured プロジェクト構造

SSO のサンプルには以下が含まれます。

- 保護しようとしているグリーティングサービスのソース
- SSO サーバーをデプロイするテンプレートファイル (**service.sso.yaml**)
- サービスのセキュリティーを保護する Keycloak アダプターの設定

### 5.6.2. Red Hat SSO デプロイメントの設定

この例の **service.sso.yaml** ファイルには、事前設定された Red Hat SSO サーバーをデプロイするすべての OpenShift 設定項目が含まれます。SSO サーバーの設定は、この演習のために簡略化されており、ユーザーとセキュリティー設定が事前に設定された、すぐに使用できる設定を提供します。**service.sso.yaml** ファイルには非常に長い行が含まれ、gedit などの一部のテキストエディターでは、このファイルの読み取りに問題がある場合があります。

**警告**

実稼働環境では、この SSO 設定を使用することは推奨されません。具体的には、セキュリティー設定の例に追加された単純化は、実稼働環境での使用に影響を及ぼします。

**表5.5 SSO サンプルの簡素化**

| 変更点 | 理由 | 推奨事項 |
| --- | --- | --- |
| デフォルト設定には、**yaml 設定ファイルに秘密鍵** と公開鍵の両方が含まれます。 | これは、エンドユーザーが Red Hat SSO モジュールをデプロイし、内部や Red Hat SSO の設定方法を知らなくても使用可能な状態にできるためです。 | 実稼働環境では、秘密鍵をソース制御に保存しないでください。サーバー管理者が追加する必要があります。 |
| 設定済みの **クライアントはコールバック URL を受け入れ** ます。 | 各ランタイムにカスタム設定を使用しないように、OAuth2 仕様で必要なコールバックの検証を回避します。 | アプリケーション固有のコールバック URL には、有効なドメイン名を指定する必要があります。 |
| **クライアントには SSL/TLS が不要であり、セキュアなアプリケーションは HTTPS で公開されません。** | サンプルは、ランタイムごとに証明書の生成を要求しないことで単純化されています。 | 実稼働環境では、セキュアなアプリケーションはプレーン HTTP ではなく HTTPS を使用する必要があります。 |
| **トークンのタイムアウトがデフォルトの 1 分から 10 分に増えました。** | コマンドラインの例を使用した場合のユーザーエクスペリエンスを向上します。 | セキュリティーの観点から、攻撃者はアクセストークンが拡張されていると推測する必要があるウィンドウ。潜在的な攻撃者が現在のトークンを推測するのがより困難になるため、このウィンドウを短くすることが推奨されます。 |

### 5.6.3. Red Hat SSO レルムモデル

**master** レルムは、この例のセキュリティーを保護するために使用されます。コマンドラインクライアントとセキュアな REST エンドポイントのモデルを提供する事前設定されたアプリケーションクライアント定義は 2 つあります。

また、Red Hat SSO **master** レルムには、**admin** および **alice** のさまざまな認証および認可の結果の検証に使用できる 2 つの事前設定されたユーザーもあります。

#### 5.6.3.1. Red Hat SSO ユーザー

セキュリティーが保護された例のレルムモデルには、次の 2 つのユーザーが含まれます。

admin

**admin** ユーザーのパスワードは **admin** で、レルム管理者です。このユーザーは Red Hat SSO 管理コンソールに完全アクセスできますが、セキュアなエンドポイントへのアクセスに必要なロールマッピングはありません。このユーザーを使用して、認証されているが承認されていないユーザーの動作を説明することができます。

alice

**alice** ユーザーには **password** のパスワードがあり、正規のアプリケーションユーザーです。このユーザーは、セキュアなエンドポイントへの認証および認可が成功したアクセスを実証します。ロールマッピングの例は、デコードされた JWT ベアラートークンにあります。

```
{
  "jti": "0073cfaa-7ed6-4326-ac07-c108d34b4f82",
  "exp": 1510162193,
  "nbf": 0,
  "iat": 1510161593,
  "iss": "https://secure-sso-sso.LOCAL_OPENSHIFT_HOSTNAME/auth/realms/master", (1)
  "aud": "demoapp",
  "sub": "c0175ccb-0892-4b31-829f-dda873815fe8",
  "typ": "Bearer",
  "azp": "demoapp",
  "nonce": "90ff5d1a-ba44-45ae-a413-50b08bf4a242",
  "auth_time": 1510161591,
  "session_state": "98efb95a-b355-43d1-996b-0abcb1304352",
  "acr": "1",
  "client_session": "5962112c-2b19-461e-8aac-84ab512d2a01",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [ (2)
      "example-admin"
    ]
  },
  "resource_access": { (3)
    "secured-example-endpoint": {
      "roles": [
        "example-admin" (4)
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "view-profile"
      ]
    }
  },
  "name": "Alice InChains",
  "preferred_username": "alice", (5)
  "given_name": "Alice",
  "family_name": "InChains",
  "email": "alice@keycloak.org"
}
```

1. **iss** フィールドは、トークンを発行する Red Hat SSO レルムインスタンス URL に対応します。トークンを検証するには、セキュアなエンドポイントデプロイメントで設定する必要があります。
2. **roles** オブジェクトは、グローバルレルムレベルでユーザーに付与されたロールを提供します。この例では、**alice** には **example-admin** ロールが付与されています。セキュリティーが保護されたエンドポイントが認可されたロールのレルムレベルを検索することがわかります。
3. **resource_access** オブジェクトには、リソース固有のロール付与が含まれます。このオブジェクトの下には、セキュアな各エンドポイントのオブジェクトを見つけます。
4. **resource_access.secured-example-endpoint.roles** オブジェクトには、**secured-example-endpoint** リソースの **alice** に付与されるロールが含まれます。
5. **preferred_username** フィールドは、アクセストークンの生成に使用されたユーザー名を提供します。

#### 5.6.3.2. アプリケーションクライアント

OAuth 2.0 仕様では、リソースの所有者の代わりにセキュアなリソースにアクセスするアプリケーションクライアントのロールを定義できます。**master** レルムには、以下のアプリケーションクライアントが定義されています。

**demoapp**

これは、アクセストークンの取得に使用されるクライアントシークレットを持つ **confidential** タイプのクライアントです。トークンには **alice** ユーザーの権限が含まれ、**alice** が Thorntail、Eclipse Vert.x、Node.js、および Spring Boot ベースの REST サンプルアプリケーションデプロイメントにアクセスできるようにします。

**secured-example-endpoint**

**secured-example-endpoint** は、関連するリソース (特に Greeting サービス) にアクセスするために **example-admin** ロールを必要とするベアラーのみのクライアントタイプです。

### 5.6.4. Node.js SSO アダプターの設定

SSO アダプターは **クライアント側**、または SSO サーバーのクライアントで、Web リソースのセキュリティーを強制するコンポーネントです。ここでは、これは Greeting サービスです。

#### Node.js コードのセキュリティーサンプルの実行

```
const express = require('express');
const Keycloak = require('keycloak-connect'); 1
const kc = new Keycloak({}); 2

const app = express();

app.use(kc.middleware()); 3

app.use('/api/greeting', kc.protect('example-admin'), callback); 4
```

1. **npm** モジュール keycloak-connect をインストールする必要があります。これは **required** です。keycloak-connect モジュールは、**express** との統合を提供する 接続ミドルウェア として機能します。

2 新しい **Keycloak** オブジェクトをインスタンス化し、空の設定オブジェクトに渡します。

3 Keycloak をミドルウェアとして使用するように **express** に指示します。

4 リソースにアクセスする前に、ユーザーが認証され、example-admin ロールの一部である必要があることを強制します。

**keycloak.jsonを使用した Keycloak アダプターのセキュリティーの実行**

```
{
  "realm": "master", 1
  "resource": "secured-example-endpoint", 2
  "realm-public-key": "...", 3
  "auth-server-url": "${env.SSO_AUTH_SERVER_URL}", 4
  "ssl-required": "external",
  "disable-trust-manager": true,
  "bearer-only": true, 5
  "use-resource-role-mappings": true
}
```

1 使用するセキュリティーレルム。

2 実際の keycloak **クライアント** の設定。

3 レルム公開鍵の PEM 形式。これは管理コンソールから取得できます。

4 Red Hat SSO サーバーのアドレス (ビルド時の補間)。

5 有効な場合、アダプターはユーザーの認証を試行しませんが、ベアラートークンのみを検証します。

Node.js コードのサンプルは Keycloak を有効にし、Greeting サービスの Web リソースエンドポイントの保護を強制します。**keycloak.json** は、Red Hat SSO と対話するようにセキュリティーアダプターを設定します。

**関連情報**

- Node.js Keycloak アダプターの詳細は、Keycloak documentation を参照してください。

## 5.6.5. Secured サンプルアプリケーションの Minishift または CDK へのデプロイメント

### 5.6.5.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

**前提条件**

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

**手順**

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

**Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

#### 5.6.5.2. Fabric8 Launcher を使用した Secured サンプルアプリケーションの作成

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL およびユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

- ブラウザーで Fabric8 Launcher URL に移動します。
- 画面の指示に従って、Node.js にサンプルを作成します。デプロイメントタイプについて尋ねられたら、**ローカルで構築して実行します**。
- 画面の指示に従ってください。
  完了したら、**Download as ZIP file**ボタンをクリックし、ファイルをハードドライブに保存します。

#### 5.6.5.3. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して Minishift または CDK でアプリケーションのサンプルを使用するには、Minishift または CDK Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Minishift または CDK URL に移動します。

2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.6.5.4. CLI クライアント oc を使用した Secured サンプルアプリケーションのデプロイメント

このセクションでは、Secured サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は、「Fabric8 Launcher を使用した Secured サンプルアプリケーションの作成」を参照してください。
- Fabric8 Launcher URL。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. サンプル ZIP ファイルから **service.sso.yaml** ファイルを使用して Red Hat SSO サーバーをデプロイします。

```
$ oc create -f service.sso.yaml
```

5. **npm** を使用して、Minishift または CDK へのデプロイメントを開始します。

```
$ npm install && npm run openshift -- \
    -d SSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath='{"https://"}
{.spec.host}{"/auth\n"}')
```

これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

### 5.6.6. Secured サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

Minishift または CDK の他に、マイナーな相違点のみを使用して OpenShift Container Platform にサンプルを作成し、デプロイできます。最も重要な違いは、OpenShift Container Platform でデプロイする前に、Minishift または CDK にサンプルアプリケーションを作成する必要があります。

**前提条件**

- Minishift または CDK を使用して作成された例。

#### 5.6.6.1. CLI クライアント oc の認証

コマンドラインクライアント **oc** を使用して OpenShift Container Platform でサンプルアプリケーションを使用するには、OpenShift Container Platform Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- OpenShift Container Platform のアカウント

**手順**

1. ブラウザーで OpenShift Container Platform URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Container Platform アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.6.6.2. CLI クライアント oc を使用した Secured サンプルアプリケーションのデプロイメント

このセクションでは、Secured サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. サンプル ZIP ファイルから **service.sso.yaml** ファイルを使用して Red Hat SSO サーバーをデプロイします。

```
$ oc create -f service.sso.yaml
```

5. **npm** を使用して、OpenShift Container Platform へのデプロイメントを開始します。

```
$ npm install && npm run openshift -- \
    -d SSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath='{"https://"}
{.spec.host}{"/auth\n"}')
```

   これらのコマンドは、不足しているモジュールの依存関係をインストールしてから、Nodeshift モジュールを使用して、サンプルアプリケーションを OpenShift にデプロイします。

### 5.6.7. Secured サンプルアプリケーションの API エンドポイントへの認証

Secured サンプルアプリケーションは、呼び出し元が認証および承認されている場合に **GET** 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。クライアントは最初に Red Hat SSO サーバーに対して認証を行い、認証手順によって返されるアクセストークンを使用して Secured サンプルアプリケーションに対して **GET** 要求を実行します。

#### 5.6.7.1. Secured サンプルアプリケーション API エンドポイントの取得

クライアントを使用して例を操作する場合は、**PROJECT_ID** サービスである Secured サンプルアプリケーションのエンドポイントを指定する必要があります。

**前提条件**

- Secured サンプルアプリケーションがデプロイされ、実行します。
- 認証された **oc** クライアント。

**手順**

1. 端末アプリケーションで、**oc get routes** コマンドを実行します。
   以下の表の出力例を以下に示します。

**例5.1 Secured エンドポイントの一覧**

| 名前 | ホスト/ポート | パス | サービス | ポート | 終了 |
|---|---|---|---|---|---|
| secure-sso | secure-sso-myproject.LOCAL_OPENSHIFT_HOSTNAME | | secure-sso | (すべて) | passthrough |
| PROJECT_ID | PROJECT_ID-myproject.LOCAL_OPENSHIFT_HOSTNAME | | PROJECT_ID | (すべて) | |
| sso | sso-myproject.LOCAL_OPENSHIFT_HOSTNAME | | sso | (すべて) | |

上記の例では、サンプルエンドポイントは **http://PROJECT_ID-myproject.LOCAL_OPENSHIFT_HOSTNAME** になります。**PROJECT_ID** は、developers.redhat.com/launch または Fabric8 Launcher ツールを使用してサンプルを生成する際に入力した名前に基づいています。

#### 5.6.7.2. コマンドラインを使用した HTTP 要求の認証

HTTP POST リクエストを Red Hat SSO サーバーに送信してトークンを要求します。以下の例では、CLI ツール jq を使用して JSON 応答からトークン値を抽出します。

**前提条件**

- セキュアなサンプルエンドポイント URL。詳細は、「Secured サンプルアプリケーション API エンドポイントの取得」 を参照してください。
- **jq** コマンドラインツール (任意)。ツールのダウンロードと、詳細な情報は、https://stedolan.github.io/jq/ を参照してください。

**手順**

1. **curl**、認証情報、**<SSO_AUTH_SERVER_URL>** でアクセストークンを要求し、**jq** コマンドを使用して応答からトークンを展開します。

```
curl -sk -X POST https://<SSO_AUTH_SERVER_URL>/auth/realms/master/protocol/openid-connect/token \
  -d grant_type=password \
  -d username=alice\
  -d password=password \
  -d client_id=demoapp \
```

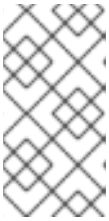
```
  -d client_secret=1daa57a2-b60e-468b-a3ac-25bd2dc2eadc \
  | jq -r '.access_token'

eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJRek1nbXhZMUhrQnpxTnR0SnkwMm5j
NTNtMGNiWDQxV1hNSTU1MFo4MGVBIn0.eyJqdGkiOiI0NDA3YTliNC04YWRhLTRlMTctOD
Q2ZS03YjI5MjMyN2RmYTIiLCJleHAiOjE1MDc3OTM3ODcsIm5iZiI6MCwiaWF0IjoxNTA3Nzkz
NzI3LCJpc3MiOiJodHRwczovL3NlY3VyZS1zc28tc3NvLWRlbW8uYXBwcy5jYWZlLWJhYmUub
3JnL2F1dGgvcmVhbG1zL21hc3RlciIsImF1ZCI6ImRlbW9hcHAiLCJzdWIiOiJjMDE3NWNjYi0w
ODkyLTRiMzEtODI5Zi1kZGE4NzM4MTVmZTgiLCJ0eXAiOiJCZWFyZXIiLCJhenAiOiJkZW1vY
XBwIiwiYXV0aF90aW1lIjowLCJzZXNzaW9uX3N0YXRlIjoiMDFjOTkzNGQtNmZmOS00NWYzL
WJkNWUtMTU4NDI5ZDZjNDczIiwiYWNyIjoiMSIsImNsaWVudF9zZXNzaW9uIjoiMzM3Yzk0MT
YtYTdlZS00ZWUzLThjZWQtODhlODI0MGJjNTAyIiwiYWxsb3dlZC1vcmlnaW5zIjpbIioiXSwicmV
hbG1fYWNjZXNzIjp7InJvbGVzIjpbImJvb3N0ZXItYWRtaW4iXX0sInJlc291cmNlX2FjY2VzcyI6ey
JzZWN1cmVkLWJvb3N0ZXItZW5kcG9pbnQiOnsicm9sZXMiOlsiYm9vc3Rlci1hZG1pbiJdfSwiY
WNjb3VudCI6eyJyb2xlcyI6WyJtYW5hZ2UtYWNjb3VudCIsInZpZXctcHJvZmlsZSJdfX0sIm5hbW
UiOiJBbGljZSBJbkNoYWlucyIsInByZWZlcnJlZF91c2VybmFtZSI6ImFsaWNlIiwiZ2l2ZW5fbmFtZ
SI6IkFsaWNlIiwiZmFtaWx5X25hbWUiOiJJbkNoYWlucyIsImVtYWlsIjoiYWxpY2VAa2V5Y2xvYW
sub3JnIn0.mjmZe37enHpigJv0BGuIitOj-
kfMLPNwYzNd3n0Ax4Nga7KpnfytGyuPSvR4KAG8rzkfBNN9klPYdy7pJEeYlfmnFUkM4EDrZY
gn4qZAznP1Wzy1RfVRdUFi0-
GqFTMPb37o5HRldZZ09QljX_j3GHnoMGXRtYW9RZN4eKkYkcz9hRwgfJoTy2CuwFqeJwZY
UyXifrfA-JoTr0UmSUed-0NMksGrtJjjPggUGS-
qOn6OgKcmN2vaVAQlxW32y53JqUXctfLQ6DhJzIMYTmOfIIPy0sgG1mG7sovQhw1xTg0vTjdx
8zQ-EJcexkj7IivRevRZsslKgqRFWs67jQAFQA
```

**<SSO_AUTH_SERVER_URL>** は **secure-sso** サービスの URL です。

**username**、**password**、**client_secret** などの属性は、通常、秘密になりますが、上記のコマンドはこの例で提供されているデフォルトの認証情報をデモの目的で使用します。

**jq** を使用してトークンを抽出しない場合は、**curl** コマンドのみを実行し、アクセストークンを手動で抽出できます。



**注記**

**-sk** オプションは curl に対し、自己署名証明書の失敗を無視します。実稼働環境ではこのオプションを使用しないでください。macOS では、**curl** バージョン **7.56.1** 以降がインストールされている必要があります。また、OpenSSL でビルドする必要があります。

1. Secured サービスを起動します。アクセス (ベアラー) トークンを HTTP ヘッダーに割り当てます。

```
$ curl -v -H "Authorization: Bearer <TOKEN>" http://<SERVICE_HOST>/api/greeting

{
   "content": "Hello, World!",
   "id": 2
}
```

**例5.2 Access (Bearer) トークンが含まれるGET 要求ヘッダーのサンプル**

```
> GET /api/greeting HTTP/1.1
> Host: <SERVICE_HOST>
> User-Agent: curl/7.51.0
```

```
> Accept: */*
> Authorization: Bearer <TOKEN>
```

**<SERVICE_HOST>** は、セキュアなサンプルエンドポイントの URL です。詳細は、「Secured サンプルアプリケーション API エンドポイントの取得」 を参照してください。

2. アクセストークンの署名を確認します。
アクセストークンは JSON Web トークン であるため、JWT デバッガー を使用してデコードできます。

   a. Web ブラウザーで、JWT デバッガー の Web サイトに移動します。

   b. **Algorithm** ドロップダウンメニューから **RS256** を選択します。

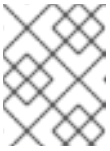
   

   **注記**

   選択後に Web フォームが更新され、正しい RSASHA256(...) 情報が Signature セクションに表示されることを確認します。そうでない場合は、HS256 に切り替えてから RS256 に戻ります。

   c. 上部のテキストボックスの **VERIFY SIGNATURE** セクションに以下のコンテンツを貼り付けます。

   ```
   -----BEGIN PUBLIC KEY-----
   MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoETnPmN55xBJjRzN/cs30OzJ
   9olkteLVNRjzdTxFOyRtS2ovDfzdhhO9XzUcTMbIsCOAZtSt8K+6yvBXypOSYvI75EUdypm
   kcK1KoptqY5KEBQ1KwhWuP7IWQ0fshUwD6jI1QWDfGxfM/h34FvEn/0tJ71xN2P8TI2Yan
   wuDZgosdobx/PAvlGREBGuk4BgmexTOkAdnFxIUQcCkiEZ2C41uCrxiS4CEe5OX91aK9
   HKZV4ZJX6vnqMHmdDnsMdO+UFtxOBYZio+a1jP4W3d7J5fGeiOaXjQCOpivKnP2yU2D
   PdWmDMyVb67l8DRA+jh0OJFKZ5H2fNgE3II59vdsRwIDAQAB
   -----END PUBLIC KEY-----
   ```

   **注記**

   これは、Secured サンプルアプリケーションの Red Hat SSO サーバーデプロイメントからのマスターレルム公開鍵です。

   d. クライアント出力から **Encoded** ボックスに **token** の出力を貼り付けます。
   **Signature Verified**記号がデバッガーページに表示されます。

#### 5.6.7.3. Web インターフェイスを使用した HTTP 要求の認証

セキュアなエンドポイントには、HTTP API の他に、対話する Web インターフェイスも含まれます。

以下の手順は、セキュリティーの実施方法、認証方法、および認証トークンの使用方法を確認するための演習です。

**前提条件**

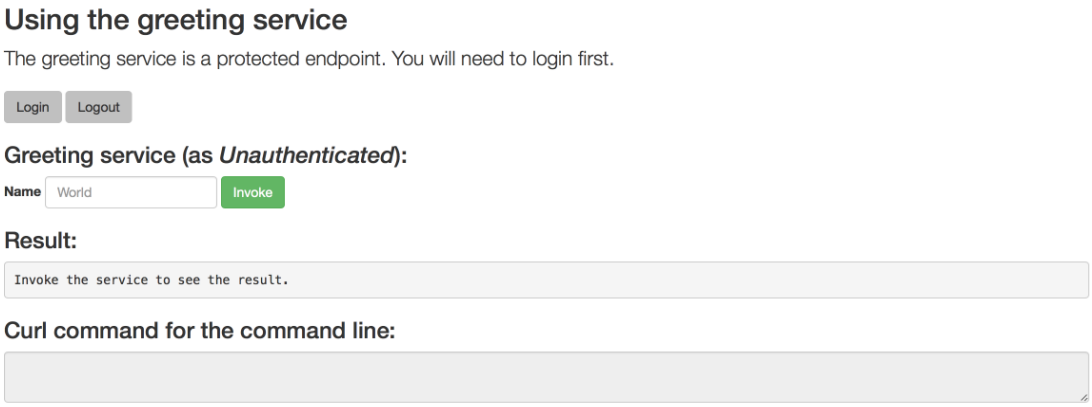
- セキュアなエンドポイント URL。詳細は、「Secured サンプルアプリケーション API エンドポイントの取得」 を参照してください。

**手順**

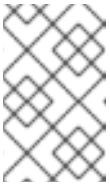
**手順**

1. Web ブラウザーで、エンドポイント URL に移動します。
2. 認証されていない要求を実行します。
   a. **Invoke** ボタンをクリックします。

**図5.1 認証されていないセキュアなサンプル Web インターフェイス**



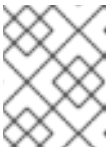

サービスは、**HTTP 403 Forbidden** ステータスコードで応答します。



**注記**

これは正しいステータスコードではありません。正しくは、**HTTP 401 Unauthorized** です。この問題は 特定されており、この例は解決され次第更新されます。

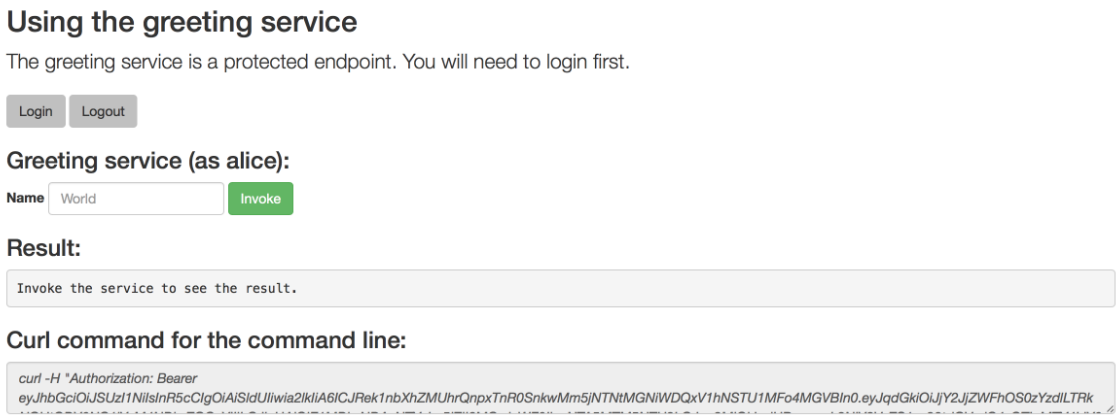
3. 認証された要求をユーザーとして実行します。
   a. **Login** ボタンをクリックして Red Hat SSO に対して認証を行います。SSO サーバーにリダイレクトされます。
   b. Alice ユーザー としてログインします。Web インターフェイスにリダイレクトされます。



**注記**

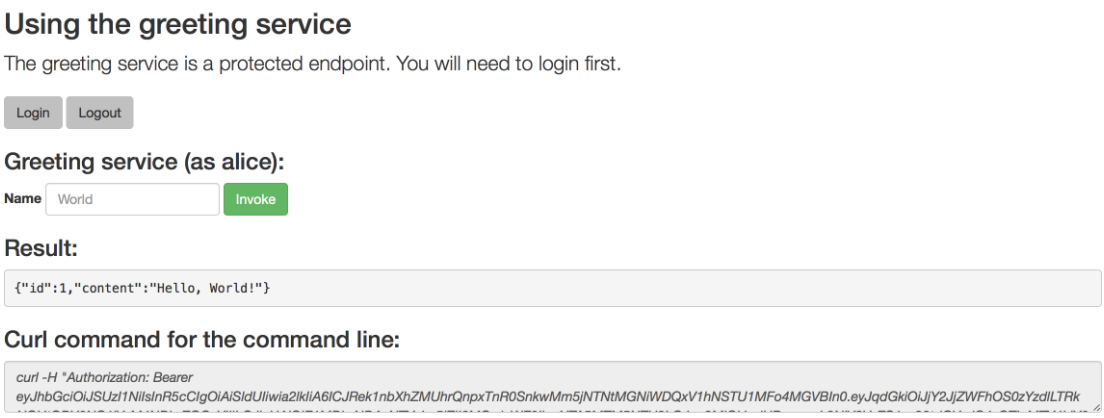
ページ下部のコマンドライン出力に、アクセス (ベアラー) トークンが表示されます。

**図5.2 (Alice として) 認証されたセキュアなサンプル Web インターフェイス**



c. 再度 **Invoke** をクリックして Greeting サービスにアクセスします。
例外がなく、JSON 応答ペイロードが表示されていることを確認します。これは、サービスがアクセス (ベアラー) トークンを受け入れ、Greeting サービスへのアクセスが承認されていることを意味します。

**図5.3 (Alice として) 認証された Greeting リクエストの結果**



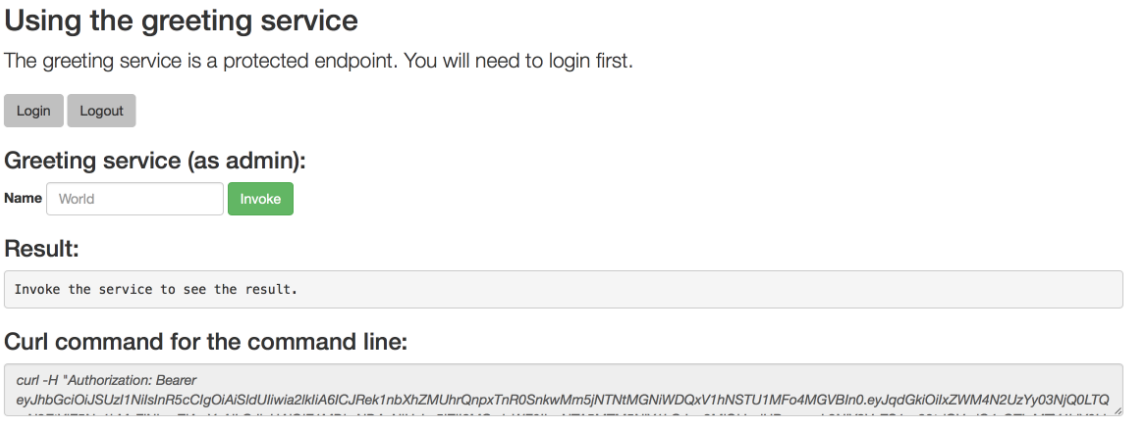

d. ログアウトします。

4. 認証された要求を管理者として実行します。

a. **Invoke** ボタンをクリックします。
これにより、認証されていない要求が Greeting サービスに送信されていることを確認します。

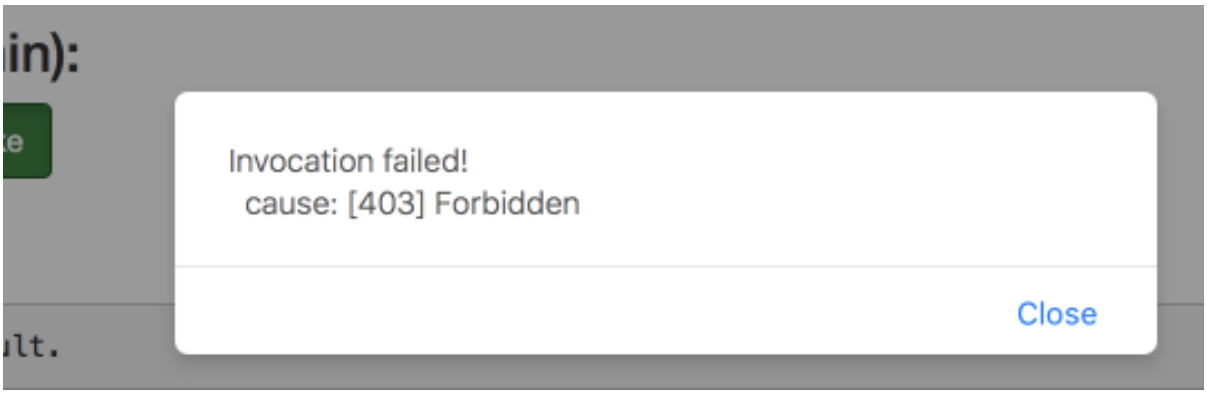
b. **Login** ボタンをクリックして admin ユーザー としてログインします。

**図5.4 (admin として) 認証されたセキュアなサンプル Web インターフェイス**




5. **Invoke** ボタンをクリックします。
このサービスは、**admin** ユーザーが Greeting サービスへのアクセスが許可されていないため、**HTTP 403 Forbidden** ステータスコードで応答します。

**図5.5 承認されていないエラーメッセージ**




### 5.6.8. セキュアな SSO リソース

OAuth2 仕様の背後にある原則に関する追加情報や、Red Hat SSO および Keycloak を使用してアプリケーションのセキュリティーを保護する方法については、以下のリンクを参照してください。

- Aaron Parecki: OAuth2 Simplified
- Red Hat SSO 7.1 ドキュメント
- Keycloak 3.2 のドキュメント
- Spring Boot のセキュアなサンプルアプリケーション
- Eclipse Vert.x におけるセキュリティー保護
- Thorntail におけるセキュリティー保護

## 5.7. NODE.JS のキャッシュの例

**重要**

以下の例は、実稼働環境での実行を目的としていません。

**制限:** このサンプルアプリケーションを Minishift または CDK で実行します。手動ワークフローを使用して、このサンプルを OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Advanced**

キャッシュサンプルは、キャッシュを使用してアプリケーションの応答時間を長くする方法を示しています。

この例では、以下の方法を示しています。

- キャッシュを OpenShift にデプロイします。
- アプリケーション内でキャッシュを使用します。

### 5.7.1. キャッシュの仕組みおよび必要なタイミング

キャッシュを使用すると、情報を保存して、一定期間アクセスできます。元のサービスを繰り返し呼び

出すよりも、キャッシュの情報により速く、またはより確実にアクセスすることができます。キャッシュを使用する欠点は、キャッシュされた情報が最新ではないことです。ただし、キャッシュに保存されている各値に **有効期限** または TTL (time to live) を設定して、この問題を減らすことができます。

**例5.3 キャッシュの例**

service1 および service2 の 2 つのアプリケーションがあるとします。

- Service1 は service2 からの値によって異なります。
  - service2 からの値が頻繁に変更されると、service1 は一定期間 service2 からの値をキャッシュすることができます。
  - キャッシュされた値を使用すると、service2 が呼び出される回数を減らすことができます。
- service1 が service2 から直接値を取得するのに 500 ミリ秒かかり、キャッシュされた値を取得するために 100 ミリ秒かかると、service1 はキャッシュされた各呼び出しにキャッシュされた値を使用することで 400 ミリ秒節約できます。
- service1 が service2 にキャッシュされていない呼び出しを 1 秒あたり 5 回 (10 秒以上) にすると、呼び出しは 50 になります。
- 代わりに service1 が 1 秒の TTL でキャッシュされた値の使用を開始した場合は、10 秒で 10 の呼び出しに削減されます。

**キャッシュサンプルの仕組み**

1. cache、cute name、および greeting サービスがデプロイされ、公開されます。
2. ユーザーは greeting サービスの Web フロントエンドにアクセスします。
3. ユーザーは、Web フロントエンドのボタンを使用して greeting HTTP API を呼び出します。
4. greeting サービスは、cute name サービスの値によって異なります。
   - greeting サービスは、最初にその値が cache サービスに保存されているかどうかを確認します。これがある場合は、キャッシュされた値が返されます。
   - 値がキャッシュされていない場合、greeting サービスは cute name サービスを呼び出し、値を返し、その値を 5 秒の TTL で cache サービスに保存します。
5. Web フロントエンドは、greeting サービスからの応答と、操作の合計時間を表示します。
6. ユーザーはサービスを複数回呼び出し、キャッシュされた操作とキャッシュされていない操作の違いを確認します。
   - キャッシュされた操作は、キャッシュされていない操作よりもはるかに高速です。
   - ユーザーは、TTL の有効期限が切れる前にキャッシュを強制的に消去することができます。

### 5.7.2. キャッシュサンプルアプリケーションの OpenShift Online へのデプロイ

以下のオプションのいずれかを使用して、OpenShift Online でキャッシュサンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.7.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.7.2.2. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- OpenShift Online のアカウント。

**手順**

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.7.2.3. CLI クライアント oc を使用した Cache サンプルアプリケーションのデプロイメント

このセクションでは、Cache サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. キャッシュサービスをデプロイします。

```
$ oc apply -f service.cache.yml
```

**注記**

x86_64 以外のアーキテクチャーを使用している場合は、YAML ファイルで Red Hat Data Grid のイメージ名をそのアーキテクチャー内の関連するイメージ名に更新します。たとえば、s390x アーキテクチャーの場合は、イメージ名を IBM Z イメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新し、ppc64le アーキテクチャーの場合は、イメージ名を IBM Power Systems のイメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新します。

5. **start-openshift.sh** を使用して OpenShift へのデプロイメントを開始します。

```
$ ./start-openshift.sh
```

6. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
cache-server-123456789-aaaaa          1/1       Running     0          8m
MY_APP_NAME-cutename-1-bbbbb       1/1       Running     0          4m
MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0          7m
MY_APP_NAME-greeting-1-ccccc       1/1       Running     0          3m
MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0          3m
```

   3 つの Pod が完全にデプロイされて起動すると、ステータスは **Running** になります。

7. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-cutename   MY_APP_NAME-cutename-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-cutename   8080
None
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-greeting   8080
None
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、greeting サービスにアクセスします。

### 5.7.3. Cache サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK でキャッシュサンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

#### 5.7.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

**前提条件**

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

**手順**

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

**Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
```

```
    https://192.168.42.152:8443

You are logged in as:
    User:     developer
    Password: developer

To login as administrator:
    oc login -u system:admin
```

#### 5.7.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Node.js でサンプルアプリケーションを作成して起動します。

#### 5.7.3.3. CLI クライアント oc の認証

**oc** コマンドラインクライアントを使用して Minishift または CDK でアプリケーションのサンプルを使用するには、Minishift または CDK Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

**前提条件**

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は、「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

**手順**

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチャルマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

#### 5.7.3.4. CLI クライアント oc を使用した Cache サンプルアプリケーションのデプロイメント

このセクションでは、Cache サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

**前提条件**

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は、「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」 を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」 を参照してください。

**手順**

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. キャッシュサービスをデプロイします。

```
$ oc apply -f service.cache.yml
```

**注記**

x86_64 以外のアーキテクチャーを使用している場合は、YAML ファイルで Red Hat Data Grid のイメージ名をそのアーキテクチャー内の関連するイメージ名に更新します。たとえば、s390x アーキテクチャーの場合は、イメージ名を IBM Z イメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新し、ppc64le アーキテクチャーの場合は、イメージ名を IBM Power Systems のイメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新します。

5. **start-openshift.sh** を使用して OpenShift へのデプロイメントを開始します。

```
$ ./start-openshift.sh
```

6. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                    READY     STATUS      RESTARTS   AGE
```

```
cache-server-123456789-aaaaa            1/1       Running     0          8m
MY_APP_NAME-cutename-1-bbbbb       1/1       Running     0          4m
MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0          7m
MY_APP_NAME-greeting-1-ccccc       1/1       Running     0          3m
MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0          3m
```

3 つの Pod が完全にデプロイされて起動すると、ステータスは **Running** になります。

7. サンプルアプリケーションをデプロイして起動すると、そのルートを決定します。

**ルート情報の例**

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-cutename   MY_APP_NAME-cutename-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME             MY_APP_NAME-cutename   8080
None
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME             MY_APP_NAME-greeting   8080
None
```

Pod のルート情報は、アクセスに使用するベース URL を提供します。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、greeting サービスにアクセスします。

### 5.7.4. Cache サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

**前提条件**

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

**手順**

- 「キャッシュサンプルアプリケーションの OpenShift Online へのデプロイ」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

### 5.7.5. 未変更の Cache サンプルアプリケーションとの対話

デフォルトの Web インターフェイスを使用して、未変更の Cache サンプルアプリケーションを操作し、頻繁にアクセスされるデータの保存方法により、サービスへのアクセスに必要な時間を短縮できます。

**前提条件**

- アプリケーションがデプロイされている。

**手順**

1. ブラウザーを使用して **greeting** サービスに移動します。

2. **サービスの起動** を一度クリックします。
   **duration** の値が **2000** を超えることに注意してください。また、キャッシュ状態が **No cached value** から **A value is cached** になっていることに注意してください。

3. 5 秒待機し、キャッシュ状態が **No cached value** に戻されるのを確認します。
   キャッシュされた値の TTL は 5 秒に設定されています。TTL の期限が切れると、値はキャッシュされなくなります。

4. **Invoke the service** をもう一度クリックして、値をキャッシュします。

5. キャッシュ状態が **A value is cached** になっている数秒の間に **Invoke the service** をさらに数回クリックします。
   キャッシュされた値を使用しているため、**duration** 値が大幅に低いことに注意してください。**Clear the cache** をクリックすると、キャッシュは強調されます。

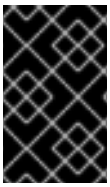
### 5.7.6. キャッシュのリソース

キャッシュに関する背景や詳細情報は、以下を参照してください。

- Spring Boot のキャッシュの例
- Eclipse Vert.x のキャッシュ
- Thorntail のキャッシュ

# 付録A NODESHIFT について

Nodeshift は、Node.js プロジェクトで OpenShift デプロイメントを実行するためのモジュールです。



**重要**

Nodeshift は、**oc** CLI クライアントがインストールされ、OpenShift クラスターにログインしていることを前提としています。また、Nodeshift は、**oc** CLI クライアントが使用している現在のプロジェクトを使用します。

Nodeshift は、プロジェクトの root にある **.nodeshift** フォルダーのリソースファイルを使用して、OpenShift Routes、Services、および DeploymentConfig の作成を処理します。Nodeshift の詳細は、Nodeshift プロジェクトページ を参照してください。

# 付録B サンプルアプリケーションのデプロイメント設定の更新

サンプルアプリケーションのデプロイメント設定には、ルート情報や readiness プローブの場所などの OpenShift でのアプリケーションのデプロイおよび実行に関連する情報が含まれます。サンプルアプリケーションのデプロイメント設定は YAML ファイルのセットに保存されます。Fabric8 Maven プラグインを使用する例では、YAML ファイルは **src/main/fabric8/** ディレクトリーにあります。Nodeshift を使用する例では、YAML ファイルは **.nodeshift** ディレクトリーにあります。

**重要**

Fabric8 Maven Plugin および Nodeshift が使用するデプロイメント設定ファイルは完全な OpenShift リソース定義である必要はありません。Fabric8 Maven Plugin と Nodeshift の両方がデプロイメント設定ファイルを取り、不足している情報を追加して完全な OpenShift リソース定義を作成できます。Fabric8 Maven Plugin によって生成されるリソース定義は **target/classes/META-INF/fabric8/** ディレクトリーにあります。Nodeshift によって生成されるリソース定義は **tmp/nodeshift/resource/** ディレクトリーにあります。

**前提条件**

- 既存のサンプルプロジェクト。
- CLI クライアント **oc** がインストールされている。

**手順**

1. 既存の YAML ファイルを編集するか、または設定を更新して追加の YAML ファイルを作成します。
   - たとえば、サンプルに **readinessProbe** が設定された YAML ファイルがすでにある場合は、**path** の値を別の利用可能なパスに変更し、readiness の有無を確認することができます。

```
spec:
  template:
    spec:
      containers:
        readinessProbe:
          httpGet:
            path: /path/to/probe
            port: 8080
            scheme: HTTP
...
```

   - **readinessProbe** が既存の YAML ファイルで設定されていない場合は、**readinessProbe** 設定を使用して新規 YAML ファイルを同じディレクトリーに作成することもできます。
2. Maven または npm を使用して、サンプルの更新バージョンをデプロイします。
3. 設定更新が、デプロイ済みのサンプルに表示されることを確認します。

```
$ oc export all --as-template='my-template'

apiVersion: template.openshift.io/v1
kind: Template
```

```
metadata:
  creationTimestamp: null
  name: my-template
objects:
- apiVersion: template.openshift.io/v1
  kind: DeploymentConfig
  ...
  spec:
    ...
    template:
      ...
      spec:
        containers:
          ...
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /path/to/different/probe
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 60
            periodSeconds: 30
            successThreshold: 1
            timeoutSeconds: 1
          ...
```

**関連情報**

Web ベースのコンソールまたは CLI クライアント **oc** を使用してアプリケーションの設定を直接更新している場合は、その変更を YAML ファイルへエクスポートして追加します。**oc export all** コマンドを使用して、デプロイされたアプリケーションの設定を表示します。

# 付録C NODESHIFT で NODE.JS アプリケーションをデプロイするための JENKINS フリースタイルプロジェクトの設定

ローカルホストの nodeshift を使用して Node.js アプリケーションをデプロイするのと同様に、Jenkins を nodeshift を使用して Node.js アプリケーションをデプロイするように設定できます。

**前提条件**

- OpenShift クラスターへのアクセス
- 同じ OpenShift クラスターで実行している Jenkins コンテナーイメージ。
- Jenkins サーバーに The Node.js プラグイン がインストールされている。
- nodeshift および Red Hat ベースイメージを使用するように設定されている Node.js アプリケーション。

  **nodeshift での Red Hat ベースイメージの使用例**

  ```
  $ nodeshift --dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12 ...
  ```

- GitHub で利用可能なアプリケーションのソース。

**手順**

1. アプリケーションの新規 OpenShift プロジェクトを作成します。
   a. OpenShift Web コンソールを開き、ログインします。
   b. **Create Project** をクリックして、新しい OpenShift プロジェクトを作成します。
   c. プロジェクト情報を入力し、**Create** をクリックします。
2. Jenkins がそのプロジェクトにアクセスできるようにします。
   たとえば、Jenkins のサービスアカウントを設定している場合は、アカウントに、アプリケーションのプロジェクトへの **edit** アクセスがあることを確認してください。
3. Jenkins サーバーで新しい フリースタイルの Jenkins プロジェクト を作成します。
   a. **New Item** をクリックします。
   b. 名前を入力し、**Freestyle プロジェクト** を選択して **OK** をクリックします。
   c. **Source Code Management** で **Git** を選択し、アプリケーションの GitHub URL を追加します。
   d. **Build Environment** で、 **Provide Node & npm bin/ folder to PATH**が確認され、Node.js 環境が設定されていることを確認してください。
   e. **Build** で、**Add build step** を選択し、**Execute Shell** を選択します。
   f. 以下を **コマンド** エリアに追加します。

```
npm install -g nodeshift
nodeshift --dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12 --
namespace=MY_PROJECT
```

**MY_PROJECT** をアプリケーションの OpenShift プロジェクトの名前に置き換えます。

g. **Save** をクリックします。

4. Jenkins プロジェクトのメインページから **Build Now** をクリックし、アプリケーションの OpenShift プロジェクトにアプリケーションのビルドおよびデプロイを確認します。
アプリケーションの OpenShift プロジェクトでルートを開いて、アプリケーションがデプロイされていることを確認することもできます。

## 次のステップ

- GITSCM ポーリング を追加すること、または **Poll SCM** ビルドトリガー を使用することを検討してください。これらのオプションにより、新規コミットが GitHub リポジトリーにプッシュされるたびにビルドを実行できます。
- Node.js プラグインを設定 する際に、nodeshift をグローバルパッケージとして追加することを検討してください。これにより、**Execute Shell** ビルドステップを追加する際に、**npm install -g nodeshift** を省略できます。
- デプロイ前にテストを実行するビルドステップを追加することを検討してください。

# 付録D PACKAGE.JSON プロパティーの内訳

nodejs-rest-http/package.json

```
{
  "name": "nodejs-rest-http",
  "version": "1.1.1",
  "author": "Red Hat, Inc.",
  "license": "Apache-2.0",
  "scripts": {
    "test": "tape test/*.js | tap-spec", 1
    "lint": "eslint test/*.js app.js bin/*",
    "prepare": "nsp check",
    "coverage": "nyc npm test",
    "coveralls": "nyc npm test && nyc report --reporter=text-lcov | coveralls",
    "ci": "npm run lint && npm run coveralls",
    "dependencyCheck": "szero . --ci",
    "release": "standard-version",
    "openshift": "nodeshift --strictSSL=false --nodeVersion=8.x", 2
    "postinstall": "license-reporter report && license-reporter save --xml licenses.xml",
    "start": "node ." 3
  },
  "main": "./bin/www", 4
  "repository": {
    "type": "git",
    "url": "git://github.com/nodeshift-starters/nodejs-rest-http.git"
  },
  "files": [ 5
    "package.json",
    "app.js",
    "public",
    "bin",
    "LICENSE",
    "licenses"
  ],
  "bugs": {
    "url": "https://github.com/nodeshift-starters/nodejs-rest-http/issues"
  },
  "homepage": "https://github.com/nodeshift-starters/nodejs-rest-http",
  "devDependencies": { 6
    "coveralls": "^3.0.0",
    "nodeshift": "^1.3.0",
    "nsp": "~3.1.0",
    "nyc": "~11.4.1",
    "standard-version": "^4.2.0",
    "supertest": "^3.0.0",
    "szero": "^1.0.0",
    "tap-spec": "~4.1.1",
    "tape": "~4.8.0",
    "xo": "~0.20.3"
  },
  "dependencies": { 7
    "body-parser": "^1.18.2",
    "debug": "^3.1.0",
```

```
    "express": "^4.16.0",
    "license-reporter": "^1.1.3"
  }
}
```

1. ユニットテストを実行するための **npm** スクリプト。**npm run test** で実行します。
2. このアプリケーションを Minishift または CDK にデプロイするための **npm** スクリプト。**npm run openshift** で実行します。**strictSSL** オプションを使用すると、自己署名証明書を使用して Minishift または CDK インスタンスにデプロイできます。
3. このアプリケーションを起動する **npm** スクリプト。**npm start** で実行します。
4. **npm start** で実行する際のアプリケーションのプライマリーエントリーポイント。
5. Minishift または CDK にアップロードされるバイナリーに追加するファイルを指定します。
6. **npm** レジストリーからインストールする開発依存関係の一覧。これらは、Minishift または CDK へのテストおよびデプロイメントに使用されます。
7. **npm** レジストリーからインストールされる依存関係の一覧。

# 付録E 追加の NODE.JS リソース

- Node.js ホームページ
- npm ホームページ

# 付録F アプリケーション開発リソース

OpenShift でのアプリケーション開発に関する詳細は、以下を参照してください。

- OpenShift インタラクティブラーニングポータル

ネットワークの負荷を削減し、アプリケーションのビルド時間を短縮するには、Minishift または CDK に Maven の Nexus ミラーを設定します。

- Maven 用の Nexus ミラーリングの設定

# 付録G SOURCE-TO-IMAGE (S2I) ビルドプロセス

Source-to-Image (S2I) は、アプリケーションソースのあるオンライン SCM リポジトリーから再現可能な Docker 形式のコンテナーイメージを生成するビルドツールです。S2I ビルドを使用すると、ビルド時間を短縮し、リソースおよびネットワークの使用量を減らし、セキュリティーを改善し、その他の多くの利点を使用して、アプリケーションの最新バージョンを実稼働に簡単に配信できます。OpenShift は、複数の ビルドストラテジーおよび入力ソース をサポートします。

詳細は、OpenShift Container Platform ドキュメントの Source-to-Image (S2I) ビルド の章を参照してください。

最終的なコンテナーイメージをアセンブルするには、S2I プロセスに 3 つの要素を指定する必要があります。

- GitHub などのオンライン SCM リポジトリーでホストされるアプリケーションソース。
- S2I Builder イメージ。アセンブルされたイメージの基盤となり、アプリケーションを実行しているエコシステムを提供します。
- 必要に応じて、S2I スクリプト によって使用される環境変数およびパラメーターを指定することもできます。

このプロセスは、S2I スクリプトで指定された指示に従ってアプリケーションソースおよび依存関係を Builder イメージに挿入し、アセンブルされたアプリケーションを実行する Docker 形式のコンテナーイメージを生成します。詳細は、OpenShift Container Platform ドキュメントの S2I ビルド要件、ビルドオプション および ビルドの仕組み を参照してください。

# 付録H 習熟度レベル

利用可能な各例は、特定の最小知識を必要とする概念について言及しています。この要件は例によって異なります。最小要件と概念は、いくつかの上達度レベルで設定されています。ここで説明するレベルの他に、各例に固有の追加情報が必要になる場合があります。

## Foundational

Foundational と評価された例は、通常、サブジェクトに関する事前の知識を必要としません。それらは、重要な要素、概念、および用語の一般的な認識およびデモンストレーションを提供します。この例の説明に直接記載されているものを除き、特別な要件はありません。

## Advanced

Advanced サンプルを使用する場合は、Kubernetes および OpenShift に加えて、例のサブジェクトエリアの一般的な概念および用語に精通していることを前提としています。サービスやアプリケーションの設定、ネットワークの管理など、独自の基本的なタスクを実行できるようにする必要があります。この例ではサービスが必要で、設定がサンプルの範囲に含まれていない場合は、適切に設定する知識があり、サービスの結果として生じる状態のみがドキュメントに記載されていることを前提とします。

## Expert

Expert サンプルは、サブジェクトに関する最高レベルの知識が必要です。機能ベースのドキュメントとマニュアルに基づいて多くのタスクを実行することが期待されており、ドキュメントは最も複雑なシナリオを対象としています。

# 付録I 用語

## I.1. 製品およびプロジェクト名

**Developer Launcher (developers.redhat.com/launch)**

Developer Launcher (developers.redhat.com/launch) は、Red Hat が提供するスタンドアロンの入門エクスペリエンスです。これは、OpenShift でのクラウドネイティブ開発の開始に役立ちます。これには、OpenShift にダウンロード、ビルド、およびデプロイできる機能のサンプルアプリケーションが含まれます。

**Minishift または CDK**

Minishift を使用してマシンで実行している OpenShift クラスター。

## I.2. DEVELOPER LAUNCHER に固有の用語

**例**

アプリケーション仕様 (例: **REST API を持つ Web サービス**)。
この例では、通常、実行する言語やプラットフォームを指定していません。説明には意図された機能のみが含まれます。

**アプリケーションの例**

特定の ランタイム に対する特定の サンプル の言語固有の実装。サンプルアプリケーションは、サンプルカタログ に記載されています。
たとえば、サンプルアプリケーションは Thorntail ランタイムを使用して実装される REST API を持つ Web サービスです。

**サンプルカタログ**

サンプルアプリケーションに関する情報が含まれる Git リポジトリー。

**ランタイム**

サンプルアプリケーション を実行するプラットフォーム。たとえば、Thorntail または Eclipse Vert.x などです。